Contract No. DAAL03-87-K-0090
June 1, 1987 - November 30, 1988

PHOENIX, A HIGH-PERFORMANCE UNIX WITH AN EMPHASIS ON DYNAMIC MODIFICATION, REAL-TIME RESPONSE AND SURVIVABILITY

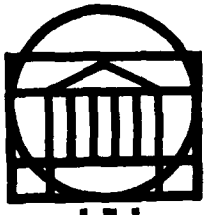
Submitted to:

U.S. Army Research Office
P.O. Box 12211
4300 S. Miami Boulevard
Research Triangle Park, NC 27709-2211

Submitted by:

R. P. Cook
Associate Professor

Report No. UVA/525186/CS89/101
December 1988



SCHOOL OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UVA/525186/CS89/101	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 24753.1-EL
6a. NAME OF PERFORMING ORGANIZATION University of Virginia Dept. of Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research Resident Representative
6c. ADDRESS (City, State, and ZIP Code) Thornton Hall Charlottesville, VA 22901	7b. ADDRESS (City, State, and ZIP Code) 818 Connecticut Ave., N. W. Eighth Floor Washington, DC 20006
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U.S. Army Research Office 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL0387-K-0090
8c. ADDRESS (City, State, and ZIP Code) P.O. Box 12211 4300 S. Miami Boulevard Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) PHOENIX, A High-Performance UNIX With An Emphasis On Dynamic Modification, Real-Time Response And Survivability	
12 PERSONAL AUTHOR(S) R. P. Cook	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM 6/1/87 TO 11/30/88	14. DATE OF REPORT (Year, Month, Day) 1988 December 2 15. PAGE COUNT 86
16. SUPPLEMENTARY NOTATION The view, opinions, and/or findings contained in this report are those of the author and should not be construed as an official department of the Army position, policy, or decision, unless so designated by other documentation.	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The goal of the Phoenix research project, which is a three year effort, is to develop a high-performance operating system for embedded applications that have a real-time response requirement. The system is to be extremely modular so that it can be easily adapted to meet different performance goals or application restrictions. Phoenix will also support a UNIX-like system call interface for compatibility with government standards. There are currently no UNIX operating systems capable of meeting "hard" real-time requirements; there are currently no UNIX operating systems that can be easily adapted to meet application requirements.

We will also investigate the problems associated with modifying an operating system and application programs remotely without halting the system. For real-time systems, the modifications must be performed in such a way that the unavailability of the system, or particular modules, is minimized.

Another aspect of the project is the analysis of operating system construction techniques that minimize the unavailability of the system when a power failure or hardware malfunction occurs and that maximize the ability of a system to "pick up" where it left off. Other areas of investigation include operating system structuring techniques, better algorithms, and better system interfaces.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Dr. David W. Hislop	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

[redacted]

Contract No. DAAL03-87-K-0090
June 1, 1987 - November 30, 1988

PHOENIX, A HIGH-PERFORMANCE UNIX WITH AN EMPHASIS ON DYNAMIC MODIFICATION, REAL-TIME RESPONSE AND SURVIVABILITY

Submitted to:

U.S. Army Research Office
P.O. Box 12211
4300 S. Miami Boulevard
Research Triangle Park, NC 27709-2211

Submitted by:

R. P. Cook
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/525186/CS89/101
December 1988

Copy No. 17

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

TABLE OF CONTENTS

LIST OF APPENDICES

THE PHOENIX PROJECT

Statement of The Problem

The goal of the Phoenix research project, which is a three year effort, is to develop a high-performance operating system for embedded applications that have a real-time response requirement. The system is to be extremely modular so that it can be easily adapted to meet different performance goals or application restrictions. Phoenix will also support a UNIX-like system call interface for compatibility with government standards. There are currently no UNIX operating systems capable of meeting "hard" real-time requirements. There are currently no UNIX operating systems that can be easily adapted to meet application requirements.

We will also investigate the problems associated with modifying an operating system and application programs remotely without halting the system. For real-time systems, the modifications must be performed in such a way that the unavailability of the system, or particular modules, is minimized.

Another aspect of the project is the analysis of operating system construction techniques that minimize the unavailability of the system when a power failure or hardware malfunction occurs and that maximize the ability of a system to "pick up" where it left off. Other areas of investigation include operating system structuring techniques, better algorithms, and better system interfaces.

Summary of Results

In the first year of the Phoenix research project, we have some notable accomplishments. First, the operating system is operational, although not all of the SVID system calls are implemented yet. The code is non-proprietary so that the operating system can be used by anyone. Also, the system has a number of unique features.

First, the implementation is layered and modular. Thus, portions of the code that are not applicable for a given embedded environment can be omitted. The system is also "open", which means that the low-level operating system interfaces are available to the application programmer, if desired. Thus, it is possible to add requests to a disk's device queue without going through the operating system. This kind of openness is an absolute requirement for certain real-time applications, but need not be used otherwise.

A second feature is the system's object-based implementation. Each process is represented as a record of dynamic size whose fields can not only be added and deleted dynamically but also whose field types can change dynamically. It took quite a bit of experimentation to derive a mechanism that had the desired flexibility while remaining cost-effective in execution time. The operating system was actually rewritten three times to experiment with different ideas. The object-based approach means that one part of the system can be easily changed without affecting other parts. This information-hiding property is essential for a system that is intended to be easily adapted to meet different system requirements.

In a traditional UNIX implementation, only one kernel process may execute at a time. In Phoenix, this restriction is eliminated; thus, the execution of a low-priority task in the kernel cannot delay the execution of a high-priority task. Furthermore, a good bit of effort was invested in minimizing lock granularity. As a result, processes accessing disjoint kernel resources will execute in parallel and will be non-interfering. A non-interference guarantee is important because it allows a real-time programmer to execute a task as a test case and then to be assured that in a production version of the system that it will complete in the same period of time, assuming that the resources that it accesses are disjoint from those used by other tasks.

In order to take advantage of the performance opportunities afforded by parallelism, two versions of the operating system were created, one for a traditional uniprocessor machine model and another for a multiprocessor machine model. Thus, Phoenix can adapt to new technology as well as new application requirements.

In a real-time system, one of the important performance criteria is the maximum interrupt latency time. This is determined by the worst-case disabled section of code in the system. In a traditional UNIX

implementation, disabling interrupts to implement critical sections is a standard practice that results in poor real-time performance. In Phoenix, the use of the DISABLE operator is limited to only two modules. The maximum interrupt latency time is bounded by the execution time of the following sequence: 1) enter READY queue; 2) remove head of READY queue; 3) perform a coroutine transfer. As a side-effect of our effort to minimize this important parameter, we also discovered a new kernel design that avoids many context switches in a multiprocessor system.

Another aspect of high-performance, real-time systems is avoiding the convoy phenomenon associated with high-traffic critical sections. We have developed an intention-based locking strategy that can ameliorate this problem.

Another feature of the Phoenix implementation is its flexible treatment of file systems. For example, it is possible to have a file system without having file names. For a real-time, embedded application, ten files might be sufficient so why should the user pay the overhead associated with directories. Similarly, a file system can be restricted to extents so that the use of index blocks can be eliminated. As the Phoenix project continues, we will be developing a library of such off-the-shelf components so that the application programmer can mix-and-match to solve their problems. We have also implemented files as objects to make it easy for applications programmers to meet special-purpose requirements.

As a final point, the most important feature of Phoenix is that it is implemented totally in a host environment using the StarLite prototyping system. The environment is portable (written in C). As a result, the operating system can be hand-tailored in a pleasant and efficient manner before attempting a conversion to a bare-machine testbed. The environment can also be used to support benchmarking and program development. For example, we can time a segment of a real-time task down to the number of instructions that it executes. This also allows us to make quantitative measurements when comparing operating system algorithms for efficiency. Finally, the environment allows us to easily share our research results with any other group without requiring them to purchase hardware identical to our own.

List of All Publications and Reports

(submitted) The StarLite Prototyping Architecture to the *Third International Conference on Architectural Support for Programming Languages and Operating Systems.*

(submitted) StarLite, A Software Prototyping Environment to *IEEE Computer.*

(in preparation) Minimizing Interrupt Latency Time in the Phoenix Operating System.

(report) An Introduction to Modula-2.

(report) An Introduction to Modular Programming.

List of All Participating Personnel

Veena Bansal, Ph.D. student

Richard Crowe, M.E., December 1987

Chris Koeritz, M.S. student

Richard McDaniel, B.S. student

Prasad Wagle, M.S. student

Jenona Whitlach, M.S., December 1987

Nancy Yeager, M.E., June 1988

APPENDIX I

MINIMIZING INTERRUPT LATENCY TIME

Minimizing Interrupt Latency Time in The Phoenix Operating System

Robert P. Cook*
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

1.0 Introduction

The Phoenix project represents an attempt to improve the technology associated with operating system construction. All software is tested using the StarLite[1] prototyping environment, which supports operating system development at the host level. That is, the Phoenix operating system executes on a virtual machine supported by an interpreter. The benefit is that new designs can be tested without resorting to the bare machine mode of development that is commonly used.

In this paper, we present the solutions to two problems that were solved as part of a recent redesign of the Phoenix kernel. The first problem was to minimize the kernel's interrupt latency, which is the time taken from the generation of an interrupt to the beginning of execution by its handler. The second problem, which will be explained in a later section, relates to "race" conditions associated with process queuing.

2.0 Minimizing Interrupt Latency

We believe that the Phoenix kernel design is optimal with respect to minimizing interrupt latency times. Furthermore, the worst case time is bounded by a constant that can be derived when the kernel is constructed. The bound is then invariant across all application programs.

There are some assumptions. First, we assume that priority-ordering is desirable. Second, only disable/enable, low-level control is assumed. Third, device synchronous operations are not allowed; that is, a user cannot disable interrupts to manipulate a device's control registers. Finally, we assume a single processor, although the technique is also valid for multiprocessors.

Before explaining the idea, we point out that minimizing the time from the occurrence of an interrupt to the initiation of its handler is not the same as minimizing the handler's completion or response time, which is a function of the availability of resources, such as locks, and the complexity of the handler's algorithm. The major impediment to minimizing interrupt latency time is the indiscriminate use of disable/enable to protect critical sections. (Brinch Hansen has named such a critical section a **monolithic monitor**.) We restrict the use of disable/enable to critical sections associated with process synchronization operations. In Phoenix, these critical sections are isolated in two kernel modules, Atomic and Run.

Since disable/enable only occur in the kernel, the kernel can be analyzed to determine the disable time of its longest critical sections. This time indicates the upper bound on the latency of external interrupts both within the kernel and for any user program.

*This work is supported by ARO under contract DAAL03-87-K0090 and by ONR under contract N00014-86-K0245.

The Atomic module implements a general semaphore type that can be used for critical sections, resource counters, or as private semaphores. When an Atomic semaphore is used to protect a critical section, the execution time within the critical section must be "short". As a result, the execution time of the entry/exit code must be minimized. Therefore, processes waiting to enter a "short" critical section are queued in FIFO rather than priority order. Figure 1 lists the interface specification for Atomic and illustrates its use to implement a critical section. Notice that Atomic uses the Run module in its implementation.

The P and V procedures have the "standard" interpretation. The Test procedure is used to determine the current state (count value) of a semaphore variable.

```
DEFINITION MODULE Atomic;
  IMPORT Run;

  TYPE Semaphore = RECORD
    count : INTEGER;
    q :  Run.ProcessQueue;
  END; (* Semaphore *)

  PROCEDURE InitSemaphore(VAR s:Semaphore;
                          count:CARDINAL);
    (* count=0 - Private Semaphore *)
    (* count=1  - Critical Section    *)
    (* count=N - Resource Counter *)
  PROCEDURE P(VAR s:Semaphore);
  PROCEDURE V(VAR s:Semaphore);
  PROCEDURE Test(VAR s:Semaphore):INTEGER;
END Atomic.


VAR s : Atomic.Semaphore;

Atomic.InitSemaphore(s, 1);

P(s);                    Disabled
(* critical-section *)   Enabled
V(s);                    Disabled
```

Figure 1. The Atomic Module

Atomic semaphores are used in other modules in the operating system to implement non-atomic semaphores, condition variables, monitors, and other synchronization abstractions, none of which use disable/enable. The implementations of P and V are straightforward except for their interactions with the Run module. Figure 2 lists the code for Atomic.P.

```
PROCEDURE P(VAR s:Semaphore);
  VAR b:BOOLEAN;
    pp:Run.pProcess;
BEGIN
  b := Traps.DISABLE(); (* save old state *)
  DEC(s.count);
  IF s.count < 0 THEN  (* block caller *)
    pp := Run.Self();
    Run.SetToStop(pp^, s.q);
    PLinks.AddBefore(s.q.tail, pp^.link); (* FIFO*)
    Run.SetStop(pp^);
  END; (* IF *)
  Traps.RESTORE(b);  (* restore old state *)
END P;
```

Figure 2. The P Operation

The P operation decrements the semaphore's count. If it becomes negative, the calling process is blocked on the semaphore's queue. The PLinks module exports a doubly-linked list type as well as delete and insert operators. PLinks does not protect its critical sections; thus, it depends on its caller to maintain the integrity of an operation. In the case of Atomic, the P operation is indivisible (i.e. disabled); therefore, the call to PLinks is indivisible. The interaction with the Run module, which is described in the next section, is the most interesting part of the implementation.

3.0 The Queuing Problem

The queuing problem occurs when a process decides to block in a queue (as in Figure 2) at any level of the system. The code sequence usually has the attributes listed in Figure 3.

First, the calling process must obtain the lock that serializes queue operations. Next, the blocking criteria is tested (e.g. "count<0" for P). If the

process must wait for the invariant to become true, the lock must be released and the process blocked.

The queuing problem results because steps three and four are not indivisible. That is, an interrupt can result in the test of a queue's status, which then causes the delayed process to be unblocked. However, there is a race condition between the process attempting to block in step four and the interrupt routine attempting to wake it. Obviously, the process must enter the queue before it can be removed.

1. Get Lock.
2. Test Blocking Criteria, Update Process Status.
3. Release Lock.
4. Process Blocks in Queue.
   Results in a context switch to another process.

Figure 3. A Typical Delay Sequence

The traditional implementation of this code sequence, which is listed in Figure 4, solves the race condition by making steps three and four indivisible. Disabling interrupts is used to protect the critical section. The benefit of disabling interrupts to effect the locking is that when the context switch to the next ready process occurs, the lock is effectively released. Each process has its disable/enable status encoded in its state vector; thus, a context switch from a disabled to an enabled process releases, or opens, the critical section. The traditional solution has the disadvantage that steps three to five can involve arbitrary delays, which is inconsistent with the goal of minimizing interrupt latency.

1. Get Lock.
2. Test Blocking Criteria, Update Process Status.
3. b := Traps.DISABLE();
4. Release Lock.
5. Process Blocks in Queue.
   Context switch can enable interrupts.
6. Traps.RESTORE(b);

Figure 4. The Traditional Solution

A second solution can be obtained by implementing a primitive that combines steps four and five into a single indivisible operation. On the surface, this might appear to be the same as the previous solution. However, Step four in Figure 4 can be an arbitrary code sequence. In a software system implemented as a module hierarchy, each level is free to define its own lock abstraction. If the release-lock and block operations are combined at a low level, then either every software layer has to use the same locking mechanism or the lowest layer has to know about every higher-layer lock type. The former is overly restrictive while the latter violates modular programming principles.

We have experimented with several other solutions that were even more unsatisfactory and will not be presented here. The current implementation of the Run module, which is presented next, represents our best solution to date.

3.1 The Run Module's Solution

The Run module implements a solution to the queuing problem that is based on a variation of intention-mode locking[2]. Before releasing the lock in Step 3 of Figure 3, a process records its intention to block itself. At this point, the "block" operation is separated into three steps (1) linking the process in a delay queue, (2) releasing the lock, and (3) actually stopping. The steps are listed in Figure 5, which you may recognize as an elaboration of the P implementation in Figure 2.

1. Get Lock.
2. Test Blocking Criteria, Update Process Status.
2a. pp := Run.Self(); (* Obtain my "tag" *)
2b. Run.SetToStop(pp^, queue); (* Intention *)
3. Process Is Linked into the Queue.
3a. Release Lock.
3b. Run.SetStop(pp^); (* Process actually stops *)
   Results in a context switch to another process.

Figure 5. Intention-Based Solution

In order to derive a solution that is correct, we must examine all possible interleavings of the block/unblock process sequence. Furthermore, the Run module's procedures are critical sections with respect to the process' state information; thus, the disable time of their implementations must be accounted for to determine the bound on latency time. To facilitate the analysis, Figure 6 lists the block/unblock code sequences. Notice that a process typically blocks itself but must be unblocked by another process. We assume that, as was the case in Figure 5, locks protect the indivisibility of the first two steps in each operation.

```
                    Block Process (A, B)

1.  Get Lock.
2.  Test Blocking Criteria, Update Status.
2a. pp := Run.Self(); (* Obtain my "tag" *)
2b. Run.SetToStop(pp^, queue); (* Intention *)        A
3.  Process Is Linked into the Queue.
3a. Release Lock.

3b. Run.SetStop(pp^); (* Process stops *)             B

                   UnBlock Process (C, D)

1.  Get Lock.
2.  Test Wakeup Criteria, Update Process Status.
2a. Process Is UnLinked from the Queue.
2b. Run.SetToRun(pp^, queue); (* Intention *)        C
3.  Release Lock.

4.  Run.SetRunning(pp^); (* Process starts *)        D
```

Figure 6. Block/UnBlock Process

The actions associated with steps A and C are indivisible subject to the integrity of the lock usage. Steps B and D are indivisible because the Run module is implemented using disable/enable for its critical sections. Therefore, the possible interleavings are ABCD, ACBD, and ACDB. We will examine each possibility in turn to produce a state-machine implementation that correctly reflects the legal transitions.

The first case, ABCD, represents the sequential execution of the two code sequences that comprise the critical section. As a result, there is no interference and no possibility of error.

In the next two cases, ACBD and ACDB, a process executes ToStop and then an interrupt occurs. The interrupt routine, which must be of higher priority than the running process to be executed at all, then initiates the ToRun action.

In the DB case, the interrupt routine continues with SetRunning, returns from the interrupt, and the original process executes the SetStop call. At this point, the appropriate action is not to stop at all, just continue execution.

For the ACBD case, the ToRun occurs before the SetStop. There are two choices. First, we could record the anticipation of the SetRunning and then let the process continue execution after the SetStop. Secondly, the process could be blocked until the SetRunning occurred. The first solution, which we adopted, avoids extra context switches. The correctness of the implementation can be guaranteed by allowing only legal state transitions as illustrated by the diagram in Figure 7 on the next page.

In the diagram, the dashed rectantages indicate a process in the "runnable" state. Such a process may, or may not, be assigned to a physical processor. The solid rectangles indicate states in which a process may not be assigned to a processor. The legal state transitions are labeled with the procedure names that are exported by the Run module. Notice that a process that is the object of the ToStop-ToRun-Running-Stop transition path remains "Runnable". In the multiprocessor version of the Phoenix kernel, the process would continue to execute.

3.2 The Solution's Benefits

The Run module's longest critical section

performs two operations: (1) add a process to a priority queue and (2) perform a context switch to the process at the head of the "Runnable" queue. There are a number of well-known techniques for minimizing the insert operation and the context-switch time is machine-dependent. Thus, the disable time is minimal, which minimizes the worst-case latency time. The two-phase solution to the queuing problem supports modular programming and eliminates the need for disabled code in any other system modules (given our assumptions).

4.0 Summary

Only two modules, Atomic and Run, in our kernel contain disabled code sequences. Any disabled code necessitated by device synchronous operations must be accounted for separately. The longest disabled sequence in the kernel is contained in the V operation, which removes the process at the head of a semaphore queue and then calls Run for the insert and switch operations. For a given architecture and implementation, the execution time for these operations provides the upper bound on interrupt response time.

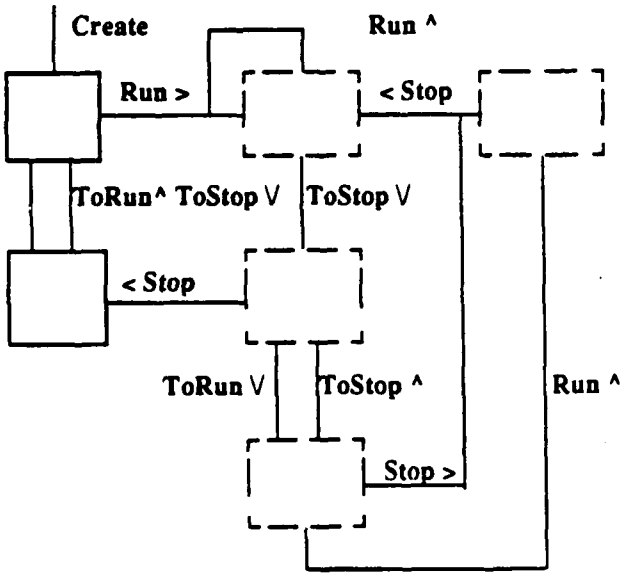


Figure 7. The Run State Diagram

Using these abstractions, we have built a UNIX kernel that is optimized for real-time response in both the uni- and multi-processor environments. For example, there is a Semaphore module, but it uses Atomic locks to protect its priority queues. Thus, while priority queue insertion in the Run module is disabled, the priority queue operations in all other modules execute with interrupts enabled.

The Phoenix brand of UNIX is an "open" operating system; that is, in addition to the standard UNIX system calls, any process can take advantage of the capabilities of any module used in the kernel's implementation.

References

[1] Cook, R.P., StarLite, A Visual Simulation Package for Software Prototyping, **Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments**, (Dec. 1986) 102-110, also **SIGPLAN Notices 22**, 1(Jan. 1987).

[2] Gray, J.N., Notes on Database Operating Systems, in **Operating Systems--An Advanced Course** (Bayer, Graham, Seegmuller eds.), Springer-Verlag 1979.

APPENDIX II

SAMPLE FILE SYSTEM DESIGNS BY STUDENTS

1. INTRODUCTION

This paper discusses the design of a fault-tolerant file system which uses a stable storage abstraction in order to achieve its fault tolerance. The layout of the disk is discussed first. Next, a discussion of error modes and assumptions follows. Then, a discussion of an on-line recovery mechanism that allows the system to be utilized while recovery proceeds is discussed. From this discussion, the maximum time to recover from the most catastrophic failure is presented.

2. WELL KNOWN OBJECT DESCRIPTORS

All objects in this file system will be implemented as files and thus each will have an object descriptor. This includes volume descriptors, directories, free lists, and *object descriptors*. Since object descriptors are to have object descriptors which in turn require object descriptors, and so on, it has been decided that the descriptors for descriptors will be implemented as a set of "well known" object descriptors which are provided to the system upon mounting a volume or upon bootup. These well known descriptors are to be kept in memory and stored either on a separate device or highly redundantly on the same device. Thus, they should always be available to the system.

3. BUDDY CYLINDERS

Stable storage copies will be placed on what will be referred to from here on out as a "buddy" cylinder. There are several design choices. If there are an even number of cylinders, we may pair cylinders n and n+1 for n in {1, 3, 5, 7, 9,...}. Thus, for example, cylinders one and two would be buddies, cylinders three and four would be buddies, and so on. If there are an odd number of cylinders, the last one would have no buddy in this scheme. We could buddy it with the next to the last cylinder adding to its share of the burden as well as requiring special handling for the last three cylinders. We could buddy up n with n+1 for n in {1, 2, 3, ...} with the last cylinder wrapping around to the first. This severely penalizes

the last cylinder in case of a failure. The goal is to have all buddies in close proximity, and the last cylinder must make a maximum seek to recover. A further problem with both of these schemes is what happens when an entire cylinder is lost? Someone loses his buddy! How do we give this cylinder a new, nearby buddy without complicating things?

To resolve these problems, we adopt the solution of cylinder groups. We don't pair the cylinders, but group them by threes or fours. Thus, we do not have the problem of what to do with the last cylinder, and many consecutive cylinders would need to be lost before we would need to regroup the cylinders. Thus, the longest seek would be three or four cylinders to recover. This is not much more of a penalty than the one required for most cylinders in the above designs. Thus, stable storage pairs in the same cylinder group will be called buddies. The location of the buddy pairs of an object are stored in the inodes for that object.

4. VOLUME DESCRIPTOR

There will be two copies of the volume descriptor which will be updated in stable storage fashion. The volume descriptors will be implemented as files. The two copies will be placed on different buddy cylinders and on different surfaces. This assures that both a cylinder and a surface may be lost while still allowing the volume descriptor to be recovered.

Having a copy of the volume descriptor on each cylinder in a known location was rejected for several reasons. First, there would be much wasted space. Second, the time required to update and validate all copies of the volume descriptor would be large. Lastly, blocks go bad dynamically. There is a placement problem if a new copy of the volume descriptor needs created (i.e. only one of the original has managed to survive). Where should this new copy be placed so that it can be easily found?

5. FREE LIST

The free list serves two purposes. First, as its name implies, it is the free list. It will also be used as a label block containing information crucial to recovery. It is a labeled bitmap with one entry for each sector. It will be implemented as a file and stable storage abstraction. The free list will be on a cylinder by cylinder basis, so one of the copies will be placed on the cylinder for which it is the free list. This may save seeks. The stable storage copy of this free list will be placed on a buddy cylinder, which will be defined in the next section. There are four pieces of information in this labeled bitmap that are used in recovery. These are the names of the two buddy inodes which are used to access this object, the address of the sector that this free list entry is for, and a flag indicating whether or not the has been checked since the last failure.

6. INODES

All inodes will be implemented as files, but not all of them need be implemented as a stable storage file. If the object is not stable, neither is the inode. Storage for each inode will begin in its own sector. The inodeDate information for all of the inodes will be compacted into one stable storage file and placed in as few sectors as possible. Thus, the information commonly written is on a different sector to help prevent failure since the more something is updated, the more likely it is to fail. It is compacted to save space. The stable storage copies of inodes and inodeDates will also be implemented as files stored on a buddy cylinder.

The index blocks pointed to in the location field (indirect and doubleIndirect blocks) will also be implemented as files. If the object and its inode are stable, then so is this file. Like all stable storage objects, its copy is to be placed on a buddy cylinder.

Inodes for files declared to not be stable will be unstable as well - a buddy inode will not be allocated. The inode and data blocks will be marked as unstable in the free list.

There will be a static number of inodes (and therefore objects) in the system. This is because they

are to be implemented as files themselves, thus requiring an inode to given for it by the boot/mount procedure. The inodes and any stable storage copies that are made may be placed anywhere on the disk. It would be advantageous to place an equal number of inodes on each cylinder. When allocating inodes, seeks may be saved by allocating one on the current cylinder. This is an advantage of having the inodes spread across cylinders. Another, more important, reason is that the loss of any one cylinder or surface cannot destroy many inodes.

7. MISCELLANEOUS

Formatting the disk requires laying out the volume descriptors, free lists, and space for inodes on the disk. The formatting program must return an inode for each of these objects since they are implemented as files scattered across the disk. These inodes must then be provided to the mount and boot operations so they may inform the system where these objects are.

When a new inode, volume descriptor, or free list must be made, a new inode must be returned for use by the boot routines as well as an indication of which of the old ones is no longer valid.

Once the link count for a certain object exceeds a certain number, the object will automatically be made stable if it is not already. In this way, we guarantee that frequently used objects or objects under heavy current use will be recoverable. If the object was unstable to begin with, it can be made unstable once again after the link count decreases sufficiently.

Storage allocation should attempt to allocate space on the current cylinder, if possible, to save seeks. The next step would be to allocate in the same cylinder group, and finally to allocate in different cylinder groups. How does this fair as the system gets full? Allocation of labeled blocks will be done using a hash function so that they are spread out and to increase the likeliness (if it is a good hash function) of finding a free labeled block quickly.

When the user creates a file or directory, he must specify whether or not it is to be stable or not. The

link counting rule helps prevent the case where the user really needed a stable storage copy and attempts to locate this situation for him. If one file in a directory is declared stable, the directory will be made stable as well.

The secondary field of a DiskAddress for an unstable object will contain a negative address. There is no need to look at this second address since the inode tells us whether it is stable or not, but this is a second line of defense.

Since volume descriptors, inodes, and free lists are implemented as files with inodes supplied at bootup, they need not be placed in labeled sectors. We can always recover them by using a recovery strategy that uses these initial inodes as well as the information in the free list. If this information is available, it would most likely be faster than looking at labels as well.

8. FAILURE MODES AND ASSUMPTIONS

This section discusses our assumptions concerning what types of failures may occur and what damage may be done by each of these failure types. Our assumptions concerning failures are as follows :

A power failure either :

- does not interrupt an operation (i.e. occurs before it starts or after it finishes)
- interrupts it in the middle in which case the object is left detectably bad
- if physical damage is done, then only one sector is damaged

Dynamic bad blocks may occur.

Head crashes may occur. All information on ONE cylinder is lost.

With these assumptions, the failure modes are as follows :

i) power failure - no damage, no interruption

ii) power failure - no damage, interruption

iii) power failure - damage, no interruption

iv) power failure - damage, interruption

v) dynamic bad block

vi) head crash - lost cylinder, no interruption

vii) head crash - lost cylinder, interruption

Power failures and head crashes must distinguish the cases where an operation was interrupted if damage was done. This is because the damage may be done to another object (for example, while moving from one stable storage copy to the other). Thus, the original operation would need completed while the damaged object needs recovered.

Several sector states need not be considered. If the object is unstable, it cannot be recovered. If the sector was free or bad, we need not worry about restoring it. The only time we need concern ourselves with this case is when damage was done to the free or bad sector while interrupting an operation on good sectors. This state is treated the same as failure modes where no damage occurs, but an operation was interrupted.

9. RECOVERY PROCEDURE

The recovery procedure is discussed in this section. While the previous section presented the assumptions and types of errors that we set out to handle, the recovery method does not make any assumptions as to what type of error occurred or what kind of damage was done. The only requirement imposed by the recovery procedure is that the system remain stable long enough for recovery to take place. Otherwise, if objects can be damaged faster than they can be recovered, there is no way to guarantee the existence of a valid copy of any object.

Upon rebooting, the system will need to validate the data that it has so it knows that the stable copies are correct. There are a number of choices. Validate the entire file system before resuming any activity, validate entries only as they are used, or some intermediate policy in which a certain number of "critical" objects are validated before resuming user activity. The correct choice depends upon the environment in which the system is to run. For the current implementation, though, an intermediate stance has been

chosen which allows users to use the system as soon as possible with file system checks taking place along with user operations. The way this is done is to validate each sector in sequence while providing FIon demandFP validation and recovery for objects requested by a user that have not already been validated. The in sequence validation takes the form of a background job which receives N% of the CPU for some suitably chosen N.

It is assumed that errors in a sector are detectable when that sector is accessed. Thus, the in sequence checks take the form of attempting to read every sector on the disk. If the read fails, the read routine calls the recovery routine outlined below to recover that sector. Eventually, every sector will have been accessed and the entire disk validated. Even if the read does not fail, the system must still insure that the object, if it was stable, matches its buddy. Thus, the read routine used in recovery will need to check this if the read is successful. If the read is unsuccessful, the recovery procedure will handle this operation.

The user's read routine will need to be able to recover sectors as well. If the user wants to access a sector that has not been validated, he should not be made to wait for the validation process to catch up to him. Recovery should take place then and there. This implies that the read operation provided to the user must be able to access the same recovery routines in case it detects an error or finds a sector that has not been verified yet. This is important even after the validation process has finished because dynamic bad blocks do occur, and it is questionable as to whether each instance of a dynamic bad block should require the entire disk to be revalidated. Thus, these read routines will be the same.

The disk access routines will need modified to perform the required checking and recovery. For example, define ReadRecover as :

```
PROCEDURE ReadRecover(FileId : CARDINAL; VAR Buffer : ADDRESS;
                      NumBytes : CARDINAL) : INTEGER;
  BEGIN
    read(FileId, Buffer, NumBytes);
    if error and stable
      then recover;
      elsif unstable then mark as bad in free list;
      elsif (labelBlock.Checked = false and labelBlock.Stable)
        then verifyCopy;
    end; (* if *)
```

```
END ReadRecover;
```

where read is the "standard" I/O routine. The other system calls would need similarly redefined. This routine would be used by both the user and the sequential validation process.

The sequential validation process works by calling ReadRecover on every sector on the free list with the exceptions being free and known bad blocks. A known bad block is a sector that was previously marked as bad and which has already been recovered if necessary.

Recovery works as outlined below. It should be noted that should an error occur while performing this recovery, that error will be recovered in on demand mode. As was already mentioned, the free list entry for each sector contains the DiskAddress of the buddy inodes which point to this sector, the address of this sector (so that it may be found in the free list), and a boolean flag that says whether the sector has been validated or not. Initially, this flag has been set to zero for every sector as the only bootup recovery procedure required.

```
PROCEDURE recover(badSector : DiskAddress);
    Open labelBlock containing this sector;
    Allocate a new sector;
    Search for the entry corresponding to the bad sector;
    Get inode locations from free list;
    Get offset of bad sector in object from free list;
    With inode1 do
       (i)  X := address in primary location at the given offset;
       (ii) Y := address in secondary location at the given offset;
       (iii) IF X = badSector
                then Copy contents at address Y to newly allocated sector;
                else Copy contents at address X to newly allocated sector;
             END; (* IF *)
       (iv) Change the appropriate address, primary or secondary, to
            be the address of the new sector;
    With inode2 do
       repeat steps (i), (ii), and (iv)
    IF damaged object was an inode
      (this information contained in free list)
       then write the new "well known" inodes to secondary store
            and update in memory.
    END; (* IF *)
    Mark free list entry for damaged sector as bad;
    Update free list entry for newly allocated sector to contain the
      appropriate information and mark as having been checked;
END recover;
```

Verify copy simply accesses the inode and validates that the two copies are identical. If they are not, the most recent one is copied into the outdated one. The free list is marked as checked.

10. RECOVERY TIME

The worst case in recovery time is when every object is stable and the entire disk must be scanned. If objects are not stable, they will be noted as being bad but no recovery is taken. Only the free list need be accessed. If every object is stable, then only half the sectors could need to have recover called on them. If more than half require this, then data has been lost. The time required for recovery is then :

(NumSectors/2) * (time to recover) +

(NumSectors) * (time to read) +

(NumSectors/2) * (time to VerifyCopy)

This assumes that all of the valid sectors come before their invalid buddies so that verifyCopy will be called for each valid sector. This verifyCopy would then initiate an on demand recovery of its invalid buddy. If recovery only receives N% of the CPU in any given time slice, validation would complete in <= (100/N) times the amount of time it would otherwise require. It is <= since it is expected that users will spend part of their time recovering in an on demand fashion.

11. OPTIMIZATIONS

When an error occurs, there may be no need to recover the entire disk sequentially if the disk controller can provide information on the location of the heads during the operation in which failure occurred. For example, if a power failure occurred and the disk controller knew that the heads were between tracks 3 and 5 on cylinder 7, then only the sectors on these affected tracks need to be recovered.

```
DEFINITION MODULE FileSys;

AddressType = RECORD
  surface, track, sector : CARDINAL;
END; (* AddressType *)

DiskAddress = RECORD
  primary, secondary : AddressType;
    (* primary = closest, secondary = farthest copies.  The secondary copy is
       only a suggestion as to where the second copy might be.  This
       information is outdated after a copy is damaged and before the address
       of the new copy can be propogated. *)
END; (* DiskAddress *)

Physical = RECORD
  revPerSec : REAL;
  blockSize : CARDINAL;       (* logical block size *)
  sectorsPerTrack,
  tracksPerCylinder,
  cylindersPerSurface,
  numSurfaces : CARDINAL;
  MaxDiskAddr : AddressType;
END; (* Physical *)

VolumeDesc = RECORD (* replicated - one on each cylinder *)
(* the volume descriptor will be implemented as a file.  The first sector(s)
   of this file will contain the tag which is the date this copy was last
   updated. *)
  version : CARDINAL;
  cylinder : CARDINAL;     (* whatcylinder do I reside on? *)
  environment : Physical;
  totalBlocks : CARDINAL; (* number of blocks in this volume *)
  numInodes : CARDINAL;   (* number of objects allowed  *)
  name : ARRAY [1..14] OF CHAR;
  readOnly : BOOLEAN;
END; (* VolumeDesc *)

ContentsType = (INDEX, INODE, DIRECTORY, FREELIST, VOLUMEDESC, DATA);

LabelType = RECORD
  inode1, inode2 : DiskAddress;  (* where the inodes may be found *)
  sectorAddress  : DiskAddress;  (* which sector is this the label for? *)
  checked,                       (* has sector been verified since last fail? *)
  free,                          (* is this sector free or in use? *)
  bad,                           (* is this a bad block? *)
  stable : BOOLEAN;              (* is this sector stable? *)
  contents : ContentsType;       (* what is it? *)
END; (* LabelType *)

FreeList = ARRAY [1..sectorsPerCylinder] OF LabelType;
    (*  The bitmap is on a per cylinder basis.  One copy of the free list
        is on the cylinder that it is the free list for.  Thus seeks may be
        minimized by finding free blocks on the same cylinder as the free
        list.  The stable storage copy of the free list is on a "buddy"
        cylinder as described in the text.

        The free list will be implemented as a file.  The first sector(s)
        of this file will contain the tag which is the date this copy was
        last updated. *)

PathElement = ARRAY [1..14] OF CHAR;  (* one element in a path name *)

Directory = RECORD
```

```
    name : ARRAY [1..MaxLevels] OF PathElement;
      (* the use of full path names allows the file system hierarchy to be
         reconstructed in case of catastrophic failure. *)
    fileId : DiskAddress;
  END; (* Directory *)

  Location = RECORD
    direct : ARRAY [1..NumDirect] OF DiskAddress;
    indirect, DoubleIndirect : DiskAddress;
  END; (* Location *)

  inode = RECORD
    (* Inodes will be implemented as files.  The first sector(s) of these files
       will contain the tag which is the date this copy was last updated. *)
    type = (FREE, DEVICE, DIRECTORY, FILE);
    stable : BOOLEAN;       (* are the object and its inode stable? *)
    uid, gid : CARDINAL;    (* user and group ids *)
    access : Access;        (* protection information *)
    linkCount : CARDINAL;   (* if > N, object made stable if not already *)
    location : Location;    (* index to blocks of object *)
    created : Date;         (* when the object was created *)
    inodeNumber : CARDINAL; (* for use in finding offset into dates block *)
  END; (* inode *)

  inodeDates = RECORD
    (* since this info is written with every access, it is separated so that
       the critical info is not accessed often.  Intuitively, the fewer times
       something is accessed, the less chance for failure.  It is to be written
       on a labeled sector. This information, for each inode, is compacted
       so that it will fit into as few sectors as possible.  In this manner,
       many dates may be lost with a failure, but there will be a stable copy.
       By compacting them in this manner, space is saved.

       InodeDates will be implemented as files.  The first sector(s) of these
       files will contain the tag which is the date this copy was last
       updated. *)

    inodeNumber : CARDINAL;  (* so we can double check that this is the right one *)
    lastAccess,
    lastModify : Date;
  END; (* inodeDates *)

  (* operations in addition to the "standard". *)

  MakeStable(name : ARRAY OF CHAR) : INTEGER;
    (* takes the name of an object and makes it stable storage if it is not
       already.  It finds a buddy inode which gets a copy of the inode
       to be made stable, and both inodes are made stable in the free list.
       The object will then be copied.  *)

  MakeUnstable(name : ARRAY OF CHAR) : INTEGER;
    (* takes the name of a file and makes it not implemented with stable
       storage.  One "buddy" inode is freed, the other is just noted as being
       unstable in the free list. *)

  Recover(name : ARRAY OF CHAR) : INTEGER;
    (* recovers the specified object from its stable storage copies *)

  (* in addition to these, create and makeNode will require an extra parameter
     to denote whether they should be stable or not at the outset.  All
     operations that alter data will need to alter both copies in a stable
     storage fashion.  *)

  END FileSys.
```

FAILURE MODES AND RECOVERY

I believe that the current state of a sector on the disk may be characterized as follows :

{free, allocated} X {open, closed} X {stable, unstable} X {labeled, unlabeled} X {normal, bad} X {clean, dirty}

This type of information will be used to characterize a sector that suffers a failure.

The allowable operations are : open, close, create, duplicate, lseek, read, write, makeNode, absoluteLink, symbolicLink, unlink, changeDirectory, labelSector, makeStable, destabilize, verify, recover.

For failure modes, we made the following assumptions,

A power failure either :

- does not interrupt an operation (i.e. occurs before it starts or after it finishes)
- interrupts it in the middle in which case the object is left detectably bad
- if physical damage is done, then only one sector is damaged

Dynamic bad blocks may occur.

Head crashes may occur. All information on ONE cylinder is lost.

With these assumptions, the failure modes are as follows :

i) power failure - no damage, no interruption

ii) power failure - no damage, interruption

iii) power failure - damage, no interruption

iv) power failure - damage, interruption

v) dynamic bad block

vi) head crash - lost cylinder, no interruption

vii) head crash - lost cylinder, interruption

Power failures and head crashes must distinguish the cases where an operation was interrupted if damage was done. This is because the damage may be done to another object (for example, while moving from one stable storage copy to the other). Thus, the original operation would need completed while the damaged

object needs recovered.

Several states need not be considered. If the object is unstable, it cannot be recovered. If the sector was bad, we need not worry about restoring it. The only time we need concern ourselves with this case is when the failure does damage to the bad sector while interrupting an operation on good sectors. This state is treated the same as failure modes on good sectors where no damage occurs, but an operation was interrupted. The type of recovery used for the assumed failure modes does not use labeled sectors. While they are useful in the case of catastrophic failure, they are not useful for sable storage. Thus, we don't care whether or not the block is labeled. Likewise, the actions taken will not depend on whether or not the object was opened or closed at the time of failure. The objective will be to return it to its previous state or to close it, which ever is appropriate for the failure. Thus, in summary

- only concerned with stable objects (can't recover unstable ones)
- failure damaging already bad sector handled in failure modes
- we don't care if a sector is labeled or not
- we don't care if an object was open or closed

The states which we must concern ourselves with are : stable X normal X {clean, dirty} X {free, allocated}. We may only dismiss one failure mode. That is when a power failure occurs causing no damage and no interruptions. Nothing needs recovering in this case. The other six failure modes must all be considered.

Of the operations, their effect on the sectors should be reflected through the current state. Thus, we need not consider operations.

The number of cases that must be considered is thus reduced to 4 X 6 = 24.

We may further reduce the number of cases by noting that when a sector is free there can be no lost data if it is damaged. Thus, the worst case for a failure concerning a free sector would be if the failure interrupted an operation concerning other sectors. This is handled the same as the case where the sector is allocated and there is no damage but the operation is interrupted. If the free sector was damaged, it must

also be marked as bad in its free lists and its buddy must be marked as unstable.

The dirty state may come about in two ways. First, if an operation is interrupted between updates to stable storage copies, then the two copies are not identical, or are dirty. This is handled in the failure modes in which an interruption has occurred. The second is when a failure has occurred which damages a copy. This is handled in the failure modes where damage occurs. The only time dirty states come about and are not already covered is when a failure occurs to an already dirty sector (i.e. before or during the recovery action for that object). In this case there are two possibilities. First, there may have been some operation on the sector that was interrupted although that sector was not damaged. Another sector may be damaged. This case is treated just like the sector was normal and some failure had occurred to make the sector inconsistent. This is only possible since there was no damage done to the sector. As far as the system is concerned, it may have just been made dirty. The second case is when damage occurs to a dirty sector. Since it is dirty, it does not have a valid stable storage copy. Thus, the object is lost. This must not occur too frequently and so we must have assurances that the expected time to failure is less than the expected recovery time. If the expected time to failure is less than the expected recovery time, stability could be gained by increasing the number of stable storage copies.

With these qualifications, we need only look at the recovery actions required for the sector state (stable, normal, clean, allocated). The others are derivable from these, as discussed above.

CASE 1 (power failure - no damage/interruption)

Since there was no damage, but the operation was interrupted, then one of the two copies was either only partially updated or not updated at all. To recover from this, one needs to replace the invalid copy with its stable storage copy.

CASE 2 (power failure - damaged sector/interruption)

Both a sector was damaged and the operation was interrupted. There are 2 cases for this.

A. The damaged sector occurred in the object of the operation. Since a sector was damaged,

it may no longer be used as a buddy in a stable storage pair. The recovery actions are :

i) allocate two free buddy sectors

ii) copy undamaged copy of sector to these new sectors

(here is where the interruption is recovered)

iii) deallocate old, undamaged sector

iv) label damaged sector as bad in free lists

v) label damaged sector's buddy as free and unstable

vi) update links to point to new sectors

B. The damaged sector occurred in an object different from that being operated on (for example, while moving from one stable storage object to its buddy).

i) recover object of operation as in CASE 1.

ii) recover damaged object, if it was stable, as in CASE 2, A.

CASE 3 (power failure - damaged sector/no interruption)

This situation is basically a dynamic bad sector. See CASE 4.

CASE 4 (dynamic bad sector)

Same as CASE 2, A

CASE 5 (head crash - no interruption)

Must recover an entire cylinder. It does not matter which cylinder since the operation either never started or already completed (i.e. it wasn't interrupted). For each sector on the lost cylinder, either it was stable or it was unstable. For stable sectors :

i) We may find the stable storage copy of the free list by looking at its "well known" inode. This locates the buddy cylinder.

ii) For each stable sector on the lost cylinder, treat it as if a dynamic bad sector had occurred and perform the steps in CASE 2, A.

CASE 6 (head crash - interruption)

We must recover the lost cylinder and, if the interrupted operation was not working with the damaged cylinder, we must recover the object of the operation as well. 2 cases

A. If the head crash occurred on the cylinder the operation was working on, then this case is just like a head crash with no interruptions. The stable copy is simply used as the good copy and recovery proceeds as in CASE 5.

B. If the head crash did not occur on the cylinder of operation, then it must have occurred between copies. To recover we must restore the object being operated on and recover the lost cylinder. We do this by

i) recover the cylinder as in CASE 5

ii) recover the object of operation as in CASE 1.

To be done :

Cases where the operation was a recovery operation. How much could be lost? (by the weekend)

For each recovery procedure, investigate what specific operations are involved and how long they may take with this design. (by the weekend)

Worst case failure and recovery time. (1 week)

Recovery time if, upon restarting the system after a failure, N% of the CPU is devoted to recovery. (?)

APPENDIX III

A SOFTWARE PROTOTYPING ENVIRONMENT

StarLite: A Software Prototyping Environment

1. Introduction

The goal of the StarLite project is to test the hypothesis that a host prototyping environment can be used to significantly accelerate the rate at which we can perform experiments in the areas of operating systems, databases, and network protocols. This paper discusses the scope of the StarLite software prototyping project.

A *software prototyping environment* is a software and/or hardware package that supports the investigation of the properties of a software system in an environment other than that of the target hardware. Prototyping tools range from IBM's Virtual Machine operating systems to discrete-event simulation languages and queuing analysis packages. Except for the VM approach to prototyping, most systems support only the analysis of an abstraction of a given software system. Thus, there is the persistent problem of validating the correctness of the model.

The StarLite software prototyping environment combines the benefits of the VM approach with those of modeling systems. The benefits of the VM approach are attained to the extent that development is in a host environment rather than on target hardware and that the same software modules are used for the host analysis and development phases, as well as for embedded testing on the target hardware.

The components of the StarLite environment include a Modula-2 compiler, a symbolic debugger, an interpreter for the prototyping architecture, a window package, and an optional simulation package. The compiler and interpreter are implemented in C for portability; the rest of the software is in Modula-2. The prototyping environment has been used to develop a non-proprietary, UNIX-like operating system that is designed for a multiprocessor architecture, as well as to perform experiments with concurrency control algorithms for distributed database systems. Both systems are organized as module hierarchies composed from reusable components.

As one measure of the effectiveness of the environment, it is often possible to fix errors in the operating system, compile, and reboot the StarLite virtual machine in less than twenty seconds. The total compilation time on a SUN 3/280 for the 66 modules (7500 lines) that comprise the operating system is 16 seconds. The StarLite interpreter, as measured by Wirth's Modula-2 benchmark program[1], executes at a speed of from one to six times that of a PDP 11/40, depending on the mix of instructions.

Another measure of the effectiveness of the prototyping environment is the ease of developing and evaluating application software. Distributed database software, which is one of the target research areas, is being developed to demonstrate StarLite's prototyping capability. The growing importance of distributed database systems in a large number of applications, such as aerospace and defense systems, industrial automation, and commercial/business applications, has been well acknowledged, and has resulted in an increased research effort in this area. However, evaluating the performance of distributed database systems or even testing new algorithms has required a large investment of time and resources. One of the primary reasons for the difficulty in successfully evaluating a distributed database system is that it takes time to develop a system. Furthermore, evaluation is complicated since it involves a large number of system parameters that may change dynamically. As a result, the field of distributed database evaluation currently lags other research areas. Performance results are inconclusive and sometimes even contradictory [2]. We feel that an important reason for this situation is that many interrelated factors affecting performance (concurrency control, buffering schemes, data distribution, etc.) have been studied as a whole, without completely understanding the overhead

imposed by each. An evaluation based on a combination of performance characterization and modeling is necessary in order to understand the impact of control algorithms on the performance of distributed database systems.

The StarLite system can reduce the time and effort necessary for evaluating new technologies and design alternatives in distributed database systems. From our past experience, we observe that a relatively small portion of a typical database system's code is affected by changes in specific control mechanisms. By properly isolating the technology-dependent portions of a database system using modular programming techniques, we can implement and evaluate design alternatives very rapidly. Although there exist tools for system development and analysis, few prototyping tools exist for distributed database experimentation. Especially if the system designer must deal with message-passing protocols and distributed data, it is essential to have an appropriate prototyping environment for success in the design and analysis tasks.

When prototyping software systems, there should be assumptions and requirements about the prototyping environment and the target system. They form the basis for evaluating prototyping tools and environments. In this paper, we first discuss assumptions and requirements for prototyping. We then present one of the programming interfaces of the StarLite environment. One of the benefits of having a prototyping environment such as StarLite is that we can develop application software that provides rapid answers to technical questions. To demonstrate the capability of the StarLite environment, distributed database systems have been implemented. The results of experiments with those systems are described.

2. Assumptions

There are three problems to address when prototyping software: intrinsic, technological, and software life-cycle problems. Our primary assumption is that the solutions to technological problems compose a relatively small percentage of a typical system's code, even though a large percentage of the design phase may be occupied with technology issues. Thus, the majority of the code deals with intrinsic problems, such as the protection mechanism for a file server, rather than with technology issues, such as the access time or capacity of a disk. By properly isolating the technology-dependent portions of a software system behind virtual machine interface definitions, software can be developed in a host environment that is separated from the target hardware.

As IBM discovered with their VM system, it is cost-effective to provide environments whose sole purpose is to support software development. A host operating system will always provide a friendlier development environment than a target hardware system. The bare machine environment is the worst possible place in which to explore new software concepts. For example, even the recovery of the event history leading up to an error in a distributed system can be a difficult and, in some cases, an impossible task.

Debugging is greatly facilitated in the host environment. The StarLite symbolic debugger supports the examination of an arbitrary number of execution "threads". As a result, the state of a distributed computation can be examined "as a whole". In addition to aiding fault isolation, the use of a host environment also facilitates fault insertion. For example, the packet error rate on a subnet could be increased to determine the effect on an internet.

Before the use of a host environment becomes feasible, however, it must be possible to simulate the machine-dependent components of a system on the host. The first step towards this goal is to require machine-independent interfaces. For instance, rather than referring to a machine status word at an absolute address, the operating system might invoke a procedure to return the word's value. When the system is executed on the host, the implementation module corresponding to the procedure would simulate the actions of the target hardware. When executing on the target, the implementation would read the content of the absolute address.

Next, for each device used in a project, it is necessary to implement a validated simulation model. This property is necessary for correctness and for the support of performance studies in which the analysis of a prototype is used to predict performance on the target hardware. The results of the virtual machine studies by Canon[3] indicate that these assumptions are reasonable. With StarLite, it is also possible to execute in a "hybrid" mode in which some modules execute on the target and some on the host. For instance, the disk for a file server could either be a target disk system or a simulated disk when the server code is executing on the host.

It is also possible to capture the timing effects of instruction execution, but not to the level of individual instructions. If that degree of accuracy is required, a VM implementation should be considered.

The final assumption is for the existence of a high-level language whose compiler supports separate compilation of units and generates reentrant code. The StarLite system can be replicated for any such language.

The separate compilation feature is used to "build" a system. For example, the Coroutines module, which is normally implemented in assembly language, forms the basis for the concurrent programming kernel, Processes. The concurrent programming kernel is then used to build the simulation package. Finally, the concurrent programming kernel, the simulation, and window packages are used to implement the virtual machine interface of the application package. Figure 1 illustrates the module dependencies for a prototyped application both in the host environment and on the target hardware. Remember that the interfaces presented to the programmer are invariant; only the implementations change with a switch to a different environment.

In order to facilitate rapid prototyping, we are developing a library of generic device objects. At present, the object library includes processes, clocks, disks, and Ethernets. Each device is presented to the user as an abstract data type, which is implemented by using the simulation package to model its characteristics and the window package to display its actions. As each data type is instantiated, a window is created to display the operations on that instance and also to serve as a point of interaction with the user. For example, a disk window provides a profile view of the device with a moving pointer to indicate head movement and sector selection. In designing the window interface, the goals were to present uniform options that could be used either in "hybrid" mode for real devices or in host-only mode. Figure 2 illustrates the StarLite windows for clocks and Ethernets.

3. Requirements for Prototyping

The primary project requirement for StarLite is that software developed in the prototyping environment must be capable of being retargeted to different architectures only by recompiling and replacing a few low-level modules. The anticipated benefits are fast prototyping times, greater sharing of software in the research community, and the ability for one research group to validate the claims of another by replicating experimental conditions exactly.

The StarLite prototyping architecture is designed to support the simultaneous execution of multiple operating systems in a single address space. For example, to prototype a distributed operating system, we might want to initiate a file server and several clients. Each virtual machine would have its own operating system and user processes. All of the code and data for all of the virtual machines would be executed as a single UNIX process. Figure 3 illustrates the machine models supported by the StarLite architecture.

In order to support this requirement, we assume the existence of high-performance workstations with large local memories. Ideally, we would prefer multi-thread support, but multiprocessor workstations are not yet widely available. We also assume that hardware details can be isolated behind high-level language interfaces to the extent that the majority of a system's software remains invariant when retargeted from the host to a target architecture.

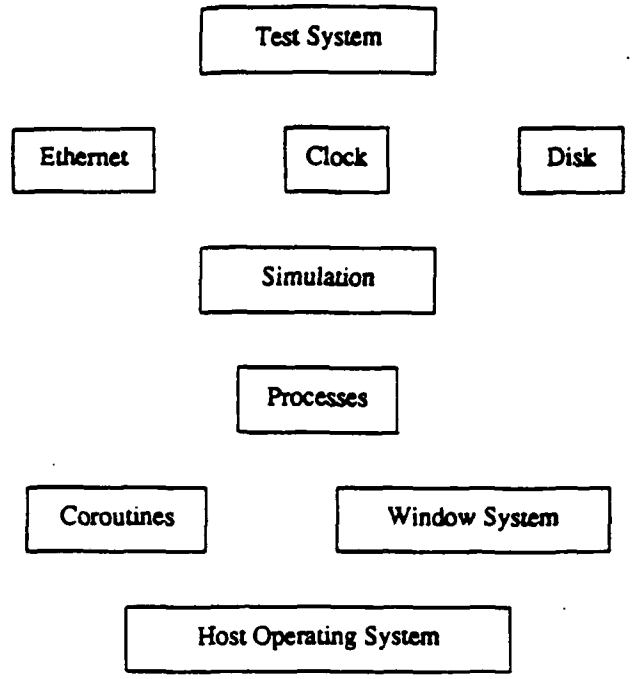


Fig. 1a. Module Hierarchy for the StarLite System

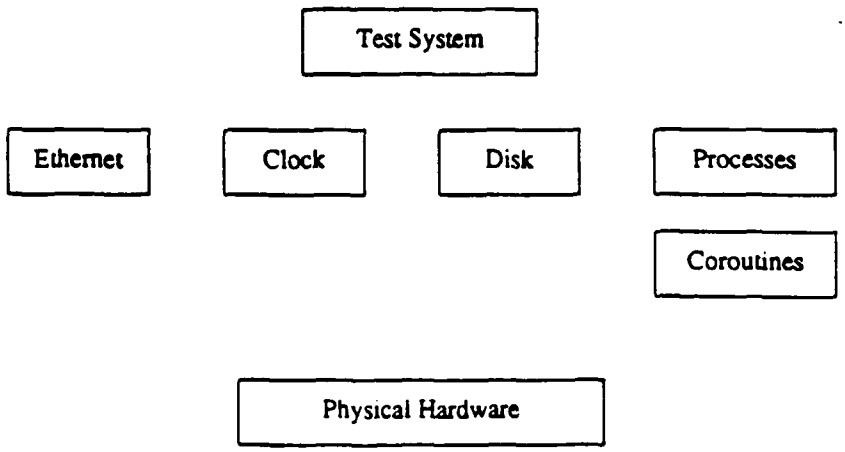


Fig. 1b. Module Hierarchy for the Target System

The architectural requirements to be satisfied by an interpreter that supports multiple operating systems running in a single, large address space are interesting. They include high speed, compact code, exception handling, good error detection, demand loading, dynamic restart, fast context switches, hybrid execution modes, and portability.

High Speed. Obviously, the speed of the host architecture is a determining factor in the usefulness of any prototyping effort. Prototyping is most effective for logic-intensive programs, such as operating systems, because the ratio of code to code-executed-per-function is high. For example, running user programs at the shell level on top of the prototype operating system, which is running on an interpreter, provides a response-level comparable (several seconds) to a PDP-11. As the number of users increase or as the number of data-intensive applications increase, the response time increases considerably. Data-intensive programs tend to apply a large percentage of their code to each data point. Thus, the number of data points determines execution speed. In many cases, having fast machines is the only effective way to prototype data-intensive applications.

Since the StarLite system uses an interpreter to define its virtual machines, we tend to stay away from data-intensive test programs. It would be nice to have an execution speed comparable to a bare machine, but that could only be achieved by building a software prototyping workstation. For now, we are satisfied as long as the edit-compile-boot-and-test cycle is significantly faster than any other environment.

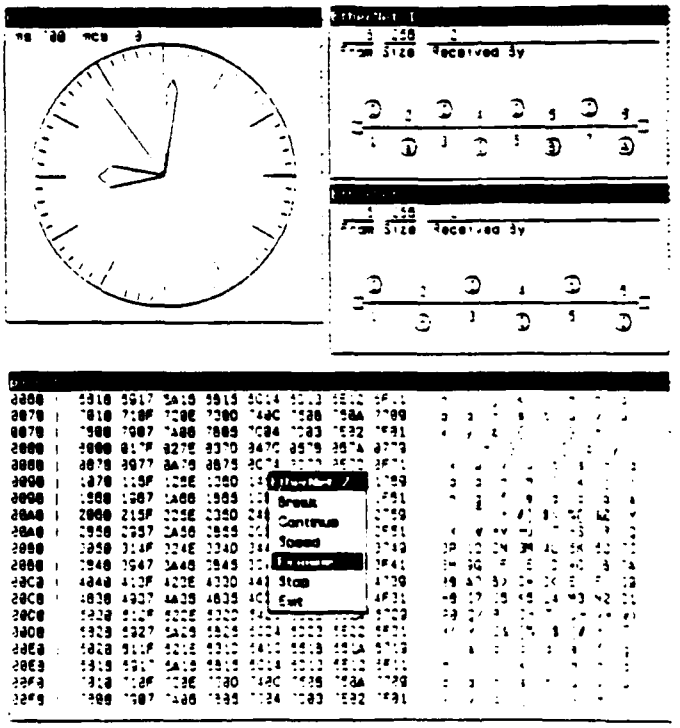


Fig. 2. Clock and Ethernet Windows

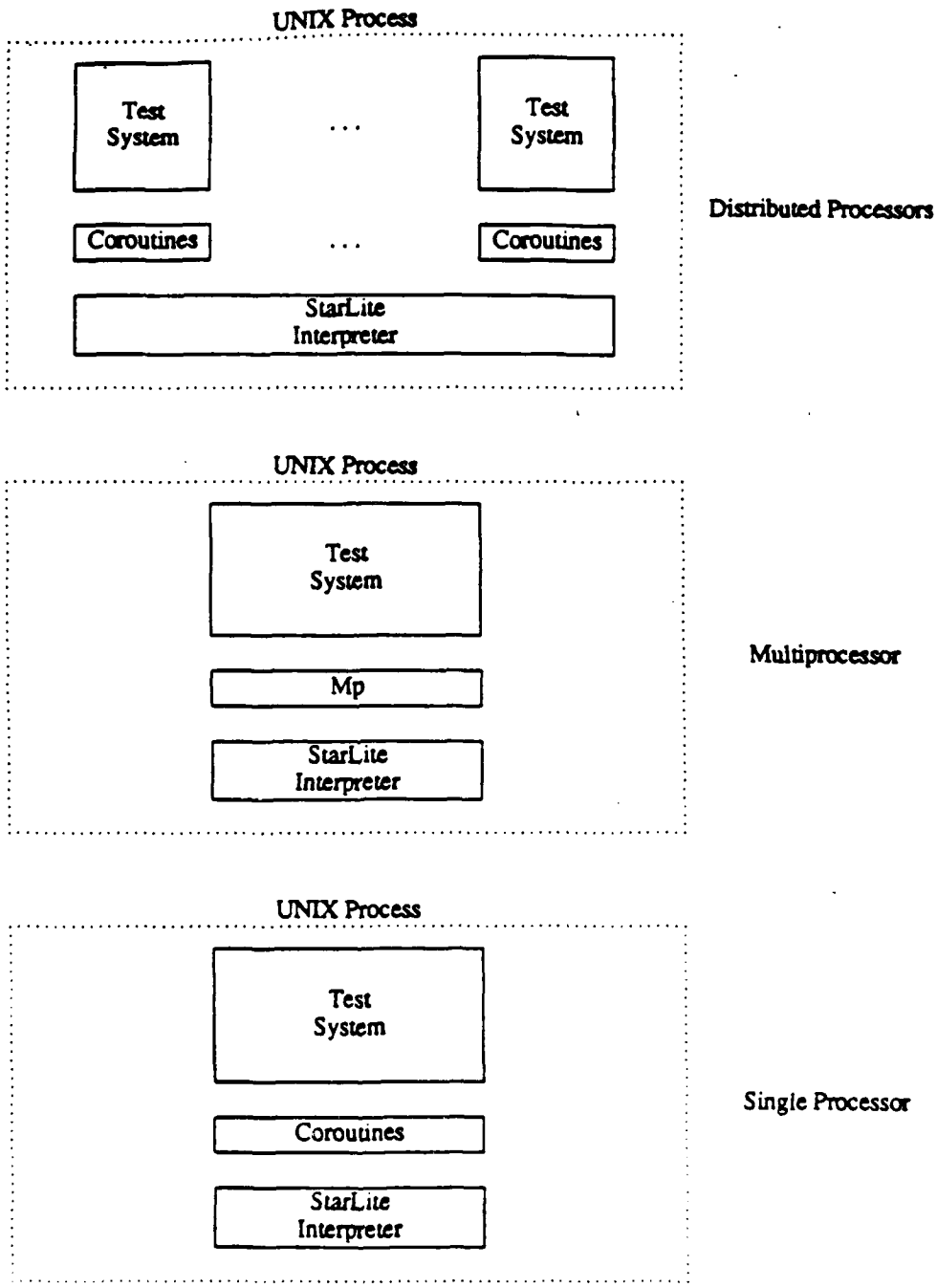


Fig. 3. Machine Models Supported by StarLite

Compact Code. The generated code for the StarLite architecture is extremely space efficient since it is based on Wirth's Lilith architecture[4]. For example, the object code (.o file) sizes for a sample 1,000 line program were SUN3-Modula2(130K), SUN3-C(65K), PC286-

C(35K), StarLite-Modula2(11K). Compact code has a significant effect on the speed with which the environment can load both system components and user-level programs that might run on those components. Compactness also increases cache locality, reduces page faults, and maximizes the quantity of software that can be co-resident in the prototyping system.

Exception Handling. The benefits of exception handling support for large system development have already been documented by Rovner[5].

> "How a program behaves in unusual situations is an essential part of its specification. Clear specification is most naturally achieved by outlining expected behavior separately from the list of the problems or unusual cases that might arise. Explicit provision in the language for decomposing a program into a normal case part and an exception handler for the exceptional cases improves predictability, robustness, and reliability."

In the StarLite environment, exception handling is provided through procedure calls to an Exception module, which is implemented in Modula-2.

Error Detection. The benefits of integrity checking as an essential component of a language's implementation have been discussed by Wirth[4].

> "Guaranteeing the validity of a language's abstractions is *not a luxury, it is a necessity*. It is as vital to inspiring confidence in a system as the correctness of its arithmetic and its memory access."

The StarLite architecture supports checks for overflow/underflow, division by zero, subrange and subscript checking, NIL pointer checks, illegal addresses, and stack overflow. Subrange and subscript checks are generated by the compiler.

Demand Loading. The StarLite architecture supports demand loading; that is, modules are loaded at the point that one of their procedures is called. Thus, a large software system begins execution very quickly and then loads only the modules that are actually referenced. For example, one version of the operating system defers loading the file system, or even the disk driver, until a file operation is performed.

At the current time, a linker is superfluous; as soon as a module is compiled, it may be executed. Demand loading and the absence of linking greatly enhances the efficacy of the StarLite debug cycle. The only limit on debugging is how fast the programmer can discover bugs and type in the changes.

Dynamic Restart. When debugging software, it can be annoying to discover an error, return to the host level, compile, and then run the system to the point of error only to discover another silly mistake. The StarLite architecture is designed so that an IMPLEMENTATION module can be compiled in a child process while the interpreter is suspended. That module can be reinserted into memory and the system restarted.

Another dynamic restart feature supports the emulation of partial failure as might be experienced in a distributed system. The Modula-2 compiler does not attempt to statically initialize any data area. Thus, any module, or set of modules, can be dynamically restarted at any time without reloading the object modules from disk. For a distributed system, the user can induce virtual processor failures and then "bring up" the operating system on those nodes without loading any software from disk.

Fast Context Switches. Unlike the "high-speed" requirement, achieving a fast context switch time can be realized independent of the characteristics of the host machine. For example,

there are no context switches within the interpreter, which is basically a C procedure in a closed loop. Therefore, a host architecture with a slow context switch time has no effect on the interpreter's context switch time: it is only a function of the state information that must be saved and restored. This is an important requirement as a typical operating system "run" can involve thousands of context switches.

Each implementation of the architecture must be balanced to match the characteristics of the host machine. The current SUN 3/280 interpreter executes 200,000 coroutine transfers/second. On the other hand, the IBM PS2/50 interpreter executes at 10,000 transfers/second.

Hybrid Execution Modes. In a prototyping environment, it is advantageous to use services that already exist in the host environment. For example, it is possible to "mount" the host file system on a leaf of a prototyped file system, or even as the prototype's "root" file system. Another example would be to use the host's database services.

Yet another example occurs in situations where the prototype would execute partially in the host and partially in a target system. An illustration of this case would be the use of a physical disk server by an operating system running in the host prototyping environment.

The keys to hybrid execution are architectural support and the definition of interfaces that remain invariant to changes in implementation technology. For example, the following interface is used in the operating system.

```
PROCEDURE Load(VAR programName : ARRAY OF CHAR):BOOLEAN;
```

It is used by the prototype operating system's "exec" system call to load user programs into memory. The interface "hides" implementation details such as the existence of a prototype file system or the virtual memory architecture. This "information hiding" principle is also used in designing device interfaces. As a result, the operating system never knows whether devices, such as disks, or services, such as "Load", are real or are emulated.

Emulation services are usually implemented by VM ROM routines. VM ROM can be used to provide functionality that the prototype software does not. A VM ROM routine has a DEFINITION module but its implementation is part of the interpreter. At execution time, the architecture intercepts calls to procedures in VM ROM and directs them to C routines. For example, when prototyping an operating system to experiment with file system issues, it is not necessary to worry about program management; VM ROM routines can be used to interface to an existing file system. At a later stage of development, the VM ROM code can be gradually replaced with code for a prototype file system.

It is easy to add additional packages to the VM ROM interface. The disadvantage is that all ROM packages must be co-resident with the interpreter. In a future version of StarLite under IBM's OS/2, all of the ROM packages will be dynamically linked on demand.

Portability. One of the benefits of developing systems in the StarLite environment is that the code can be shared with other researchers. To facilitate sharing at the object code level, the instructions generated by the compiler and its object module format are canonical. That is, the byte ordering is fixed, as is the character code (ASCII), and the floating point format (IEEE). If the host has different conventions, the compiler performs the conversions as it generates code. To the extent that an implementation module is machine invariant, it should be possible to transmit object modules from one site to another and to have them work.

The StarLite operating system design project is experimenting with the use of "safe"[5], canonical object modules for user-specified line and protocol filters, schedulers, and application-

specific file systems. For example, the operating system stores method descriptions for file access in the canonical object code format. The advantage of a canonical representation is that the volume can be transported to a different machine, which can then interpret the access method to manipulate the volume.

4. The Programming Interface

In this section, we discuss two different interfaces, one at the machine level and one at the user level, to illustrate how an interface definition can be used to "hide" environmental concerns to the extent that prototyped software can be easily retargeted from one machine to another. The first interface to be presented defines a multiprocessor; the second is for transaction management in a database system.

4.1 A multiprocessor interface

According to Wirth[1], the coroutine, or thread, is the fundamental data type for constructing multiprogramming systems. Therefore, it is a concept that remains invariant across different hardware configurations. As a result, the Coroutine type, as well as the InitCoroutine and Transfer procedures, are inherited from Wirth's[1] COROUTINE module. The Transfer procedure is an intra-processor operator; it only performs a context switch between two coroutines executing on the same processor.

However, these operators do not form a functionally complete set for a multiprocessor. It must be possible to assign a coroutine to an idle processor and to stop a coroutine that is running on another processor. The Status operator can be used to query the state of a processor. It is used by the operating system to determine how many processors are available at boot time. By convention, the bootstrap coroutine is assigned to processor zero.

The SpinLock data type and operators provide a memory synchronous test-and-set action. Spin locks are used to implement low-level, non-blocking critical sections. Interrupts and trap handling are provided by a separate interface, which is not listed. The StarLite virtual machine multiplexes the virtual processors in such a way that the effect is equivalent to executing a program on a physical multiprocessor.

The listing for the Mp module follows, together with a multiprocessor test program and its sample output. The test program, which is booted on processor zero, creates three coroutines. Each coroutine has a distinct stack for procedure variables but shares any module global variables. The code for the three coroutines is shared and its action is to continuously print the identification number for its coroutine. For the example, the identification number is the same as each coroutine's processor number. The sample output illustrates the non-deterministic nature of the execution sequence.

The Mp module has been used to execute a seven processor version of our multiprocessor operating system. The available memory and swap space on our Sun workstation is the limiting factor to running even higher number of processors. Apparently, the windows used for each machine's virtual terminals consume most of the space.

```
DEFINITION MODULE Mp; (********** The Interface for a Multiprocessor **********)

FROM SYSTEM IMPORT ADDRESS;
IMPORT COROUTINE;

CONST  NOTAVAILABLE = 0;  (**************** Processor states ****************)
       IDLE = 1;
       RUNNING = 2;
       SPINNING = 3;

TYPE SpinLock;
(* A variable of type SPINLOCK is used to "protect" critical sections.        *)

PROCEDURE SpinInit(VAR s:SpinLock);
(* This routine initializes a SPINLOCK prior to its use.                      *)

PROCEDURE SpinIn(VAR s:SpinLock);
(* This routine is executed prior to entering a critical section.             *)
(* If the critical section is available, the processor enters and is          *)
(* guaranteed exclusive access. If the critical section is not available,     *)
(* the processor spins until the SPINLOCK allows entry.                       *)

PROCEDURE SpinOut(VAR s:SpinLock);
(* This routine is executed on leaving a critical section.                    *)
(* If one or more other processors are SPINNING, one of them is chosen to enter.*)
(* The others continue SPINNING.                                              *)

TYPE Coroutine = COROUTINE.Coroutine;
(* A variable of type COROUTINE is used to identify a "thread".               *)
(* This variable is set by the INITCOROUTINE procedure when the coroutine     *)
(* is initialized. It may then be used to reference the coroutine             *)
(* when performing a TRANSFER operation.                                      *)

PROCEDURE InitCoroutine(p : PROC; stack : ADDRESS; size : CARDINAL;
                        VAR coroutine : Coroutine);
(* This routine creates a coroutine starting at procedure "p" with stack      *)
(* at address "stack" and of size "size". The coroutine identifier for        *)
(* the newly created coroutine is returned in the variable "coroutine".       *)

PROCEDURE Transfer(VAR from, to : Coroutine);
(* This routine just implements a simple context switch from                  *)
(* coroutine "from" to coroutine "to".                                        *)

PROCEDURE Begin(processor : CARDINAL; VAR to : Coroutine);
(* This routine assigns a coroutine to an IDLE processor.                     *)

PROCEDURE Stop(processor : CARDINAL; VAR from : Coroutine);
(* This routine returns a processor to the IDLE state.                        *)

PROCEDURE Status(processor : CARDINAL):CARDINAL;
(* This routine returns the status (NOTAVAIL, IDLE, RUNNING, SPINNING)        *)
(* of the selected processor.                                                 *)

END Mp.
```

A Multiprocessor Test Program

```
MODULE test;
IMPORT Mp;
FROM SYSTEM IMPORT ADR;
IMPORT InOut, Sequence;

VAR prc1, prc2, prc3 : Mp.Coroutine;
  p1Stk, p2Stk, p3Stk : ARRAY [1..500] OF CARDINAL;

PROCEDURE codeForEach(); (* this code is shared by all the processors          *)
 VAR i, loopCnt : CARDINAL; (* but the local data is unique per processor      *)
BEGIN
 i := Sequence.Count(); (* returns the next number in a sequence, a processor id *)
 loopCnt := 0;
 LOOP
  InOut.WriteCard(i, 1);
  IF loopCnt MOD 50 = 0 THEN InOut.WriteLn(); END;
  INC(loopCnt);
 END;
END codeForEach;

VAR c : CARDINAL;
BEGIN
 Mp.InitCoroutine(codeForEach, ADR(p1Stk), SIZE(p1Stk), prc1);
 Mp.InitCoroutine(codeForEach, ADR(p2Stk), SIZE(p2Stk), prc2);
 Mp.InitCoroutine(codeForEach, ADR(p3Stk), SIZE(p3Stk), prc3);
 Mp.Begin(1, prc1);      (* start prc1 on processor 1 *)
 Mp.Begin(2, prc2);      (* start prc2 on processor 2 *)
 c := 0;
 LOOP
  InOut.Write("0");
  INC(c);
  IF c MOD 50 = 0 THEN InOut.WriteLn(); END
  IF (c MOD 10)=1 THEN Mp.Stop(1, prc1); Mp.Begin(1, prc3); (* multiplex prc1/3 on processor 1 *)
  ELSIF (c MOD 10)=8 THEN Mp.Stop(1, prc3); Mp.Begin(1, prc1);
  END;
 END;
END test.
```

Sample Output for Four Processors

```
02
22222222222222222222222222222222222200000002222222222222
22222222222222222222222220003
33333333333333333333333333333332222222222222222222222222
22222222222222222200000001
11111111111111111111111111111111222222222222222222222222222222
22222222222000000333333333333333333
33333333333333333333333332222222222222222222222222222222222222
22222000001111111111111
11111111111111111111111111111122222222222222222222222222222222
22222000000333333333333333333333333
```

4.2 A distributed transaction management interface

The transaction management interface of the StarLite prototyping environment is designed to facilitate easy extensions and modifications. Server processes can be created, relocated, and new implementations of server processes can be dynamically substituted. It efficiently supports a

spectrum of distributed database functions at the operating system level, and facilitates the construction of multiple "views" with different characteristics. For experimentation, system functionality can be adjusted according to application-dependent requirements without much overhead for new system setup. Since one of the design goals of the StarLite system is to conduct an empirical evaluation of the design and implementation of application software for operating system, communication protocols, and transaction management, it has built-in support for performance measurement of both elapsed time and blocked time for each transaction[6].

The transaction management prototype provides support for concurrent multi-transaction execution, including transparency to concurrent access, data distribution, and atomicity. An instance of the prototyping environment can manage any number of virtual sites specified by the user. Modules that implement transaction processing are decomposed into several server processes, and they communicate among themselves through ports. The clean interface between server processes simplifies incorporating new algorithms into the prototyping environment, or testing alternate implementations of algorithms. To permit concurrent transactions on a single site, there is a separate process for each transaction that coordinates with other server processes. Figure 4 illustrates the structure of the transaction management prototyping environment.

The User Interface (UI) is a front-end invoked when the prototyping environment begins. UI is menu-driven, and designed to be flexible in allowing users to experiment various configurations with different system parameters. A user can specify the following:

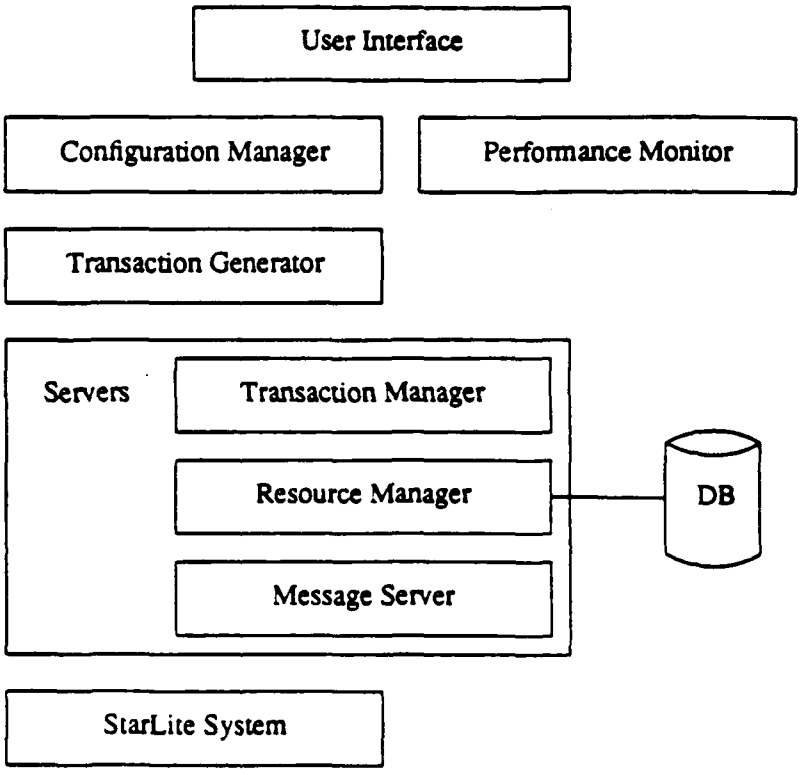


Fig. 4. Structure of the Transaction Management Prototyping Environment

- **system configuration**: number of sites and the number of server processes at each site.
- **database configuration**: database at each site with user defined structure, size, granularity, and levels of replication.
- **load characteristics**: number of transactions to be executed, size of their read-sets and write-sets, transaction types (read-only or update) and their priorities, and the mean interarrival time of transactions.
- concurrency control: locking, timestamp ordering, and priority-based.
- failure characteristics: the site and the time of a crash, and the type of recovery to be performed.

UI initiates the Configuration Manager (CM) that initializes the data structures necessary for transaction processing from user specifications. CM invokes the Transaction Generator at appropriate time intervals to generate the next transaction to form a Poisson distribution of transaction arrival times. When a transaction is generated, it is assigned an identifier that is unique among all transactions in the system. Transaction execution consists of read and write operations. Each read or write operation is preceded by an access request sent to the Resource Manager, which maintains the local database at each site. If the access request cannot be granted, the Transaction Manager (TM) executes either a blocking operation to wait until the data object can be accessed, or an abort procedure, depending on the situation. Transactions commit in two phases. The first commit phase consists of at least one round of messages to determine if the transaction can be globally committed. Additional rounds may be used to handle potential failures. The second commit phase causes the data objects to be written to the database for successful transactions. TM executes the two commit phases to ensure that a transaction commits or aborts globally.

The Message Server (MS) is a process listening on a well-known port for messages from remote sites. When a message is sent to a remote site, it is placed on the message queue of the destination site and the sender blocks itself on a private semaphore until the message is retrieved by MS. If the receiving site is not operational, a time-out mechanism will unblock the sender process. When MS retrieves a message, it wakes the sender process and forwards the message to the proper servers or TM. The prototyping environment implements Ada-style rendezvous (synchronous) as well as asynchronous message passing. Inter-process communication within a site does not go through the Message Server; processes send and receive messages directly through their associated ports.

The inter-process communication structure is designed to provide a simple and flexible interface to the client processes of the application software, independent from the low-level hardware configurations. It is split into three levels of hierarchy, as shown in Figure 5.

The Transport layer is the interface to the application software, thus it is designed to be as abstract as possible in order to support different port structures and various message types. In addition, application level processes need not know the details of the destination device. The invariant built into the design of the inter-process communication interface is that the application level sender allocates the space for a message, and the receiver deallocates it. Thus, it is irrelevant whether or not the sender and receiver share memory space, i.e., whether or not the Physical layer on the sender's side copies the message into a buffer and deallocates it at the sender's site, and the Physical layer at the receiver's site allocates space for the message. This enables prototyping distributed systems or multiprocessors with no shared memory, as well as multiprocessors with shared memory space. When prototyping the latter, only addresses need to be passed in messages without intermediate allocation and deallocation.

The Physical layer of message passing simulates the physical sending and receiving of bits over a communication medium, i.e., it is for intersite message passing. The device number in the

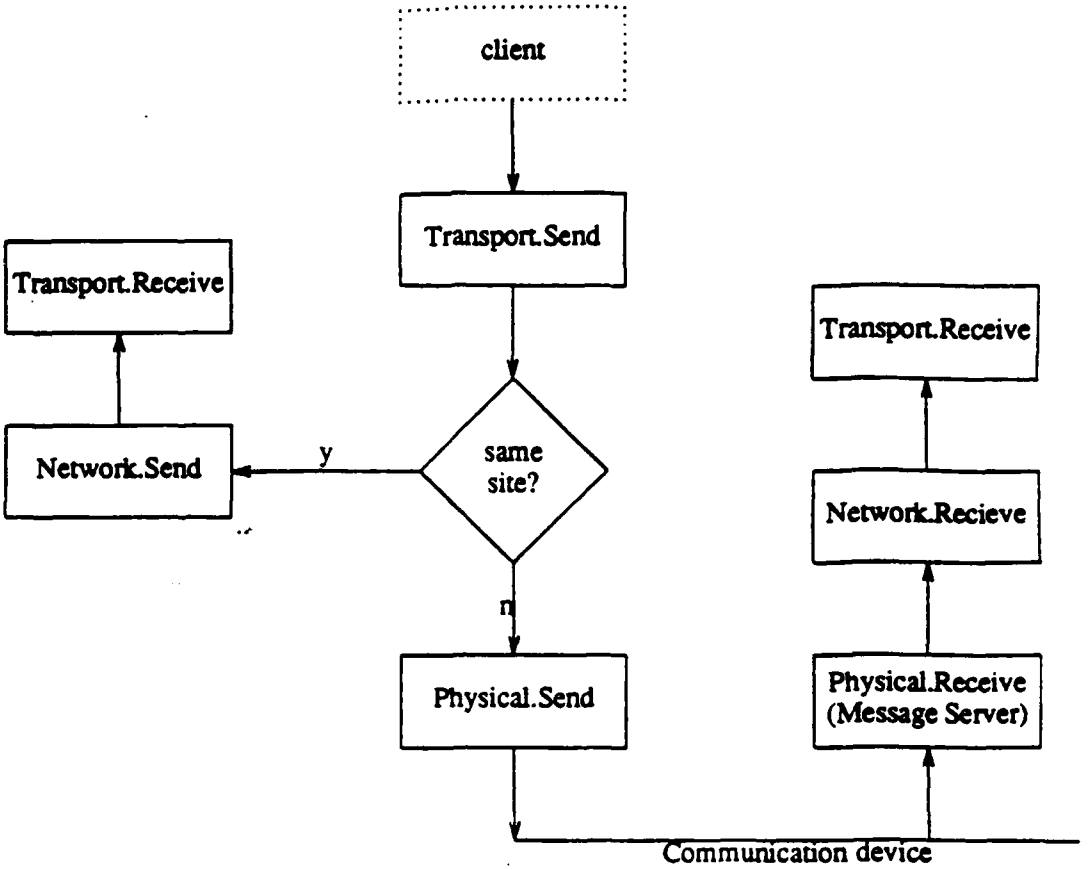


Fig. 5. Inter-Process Communication

interface is simply a cardinal number: this enables the implementation to be simple and extensible enough to support any application. To simulate sending or to actually send over an Ethernet in the target system, for example, a module could map network addresses onto cardinals. To send from one processor to another in a multiprocessor or distributed system, the cardinals can represent processor numbers.

Messages are passed to specific processes at specific sites in the Network layer of the communications interface. This layer serves to separate the Transport and the Physical layers, so that the Transport layer interface can be processor- and process-independent and the Physical layer interface need be concerned only with the sending of bits from one site to another. The Transport layer interface of the communication subsystem is implemented in the Transport module. A Transport-level Send is made to an abstraction called a *PortTag*. This abstraction is advantageous because the implementation (i.e., what a PortTag represents) is hidden in the Ports module. Thus the PortTag can be mapped onto any port structure or the reception points of any other message passing system. The Transport-level Send operation builds a packet consisting of the sender's PortTag, used for replies, the destination PortTag, and the address of the message. It then retrieves from the destination PortTag the destination device number. If this number is the same as the sender's, the Send is an intra-site message communication, and hence the middle-level Send is performed. Otherwise the send requires the Physical module for intersite

communication. Note that accesses to the implementation details of the PortTag are restricted to the module that actually implements it; this enables changing the implementation without recompiling the rest of the system. As shown in Figure 6, the Transport module, which is the highest-level interface of the inter-process communication structure for the client processes, is very simple and elegant, and it achieves the desired flexibility.

5. Prototyping a Multiversion Database: An Experiment

To evaluate the prototyping environment, we have implemented a multiversion database system and its corresponding single-version database system using the environment, and performed a series of experiments for performance comparison between them. The experiments were focused on a sensitivity study of the key parameters that affect performance, such as set size, transaction read/write ratio, interarrival time, and the database size.

In a multiversion database system, each data object consists of a number of consecutive versions. The objective of using multiple versions is to increase the degree of concurrency and to reduce the possibility of rejecting user requests by providing a succession of views of data objects. One of the reasons for rejecting a user request is that its operations cannot be serviced by the system. For example, a read operation has to be rejected if the value of data object it was supposed to read has already been overwritten by some other user request. Such rejections can be avoided by keeping old versions of each data object so that an appropriate old value can be given to a tardy read operation. In a system with multiple versions of data, each write operation on a data object produces a new version instead of overwriting it. Hence, for each read operation, the system selects an appropriate version to read, enjoying the flexibility in controlling the order of read and write operations. When a new version is created, it is *uncertified*. Uncertified versions are prohibited from being read by other transactions to guarantee cascaded-abort free[7]. A version is *certified* at the commit time of the transaction that generated the version. The

```
DEFINITION MODULE Transport;

FROM SYSTEM IMPORT BYTE, ADDRESS;
FROM Ports IMPORT PortTag;

PROCEDURE Send(pt : PortTag; VAR mess : ARRAY OF BYTE);
(* This Send is the highest-level, the transaction's interface.          *)
(* Send inverts PortTag to <processor number, Port number>.              *)
(* If the processor number is different from the sender's,               *)
(* Physical level Send is called, which sends to a different site.       *)

PROCEDURE Receive(pt : PortTag) : ADDRESS;
(* This routine is called by the destination process.                    *)
(* If the message has not arrived at the port associated with the PortTag,  *)
(* the destination process will be blocked.                              *)
(* When a message arrives at that port, this routine returns             *)
(* the address of the message.                                           *)

PROCEDURE NonblockReceive(pt : PortTag) : ADDRESS;
(* This routine is a non-blocking version of the Receive explained above.  *)
(* The destination process is not blocked if there is no message.        *)

END Transport.
```

Fig. 6. Transport Module

multiversion database system we have implemented is based on timestamp ordering. Each data object is represented as a list of versions, and each version is associated with timestamps for its creation and the latest read, and a valid bit to specify whether the version is certified.

Each transaction consists of read and write requests for data objects. Read requests are never rejected in a multi-version database system if all the versions are retained. A read operation does not necessarily read the latest committed version of a data object. A read request is transformed to a version-read operation by selecting an appropriate version to read. The timestamp of a read request is compared with the write-timestamp of the highest available version. When a read request with timestamp T is sent to the Resource Manager, the version of a data object with the largest timestamp less than T is selected as the value to be returned. The timestamp of a write request is compared with the read timestamp of the highest version of the data object. A new version with the timestamp greater than the read-timestamp of the highest certified version is built on the upper level, with the valid bit reset to indicate that the new version is not certified yet. In order to simplify the concurrency control mechanism, we allow only one temporary version for each data object. Inserting a new version in the middle of existing valid versions is not allowed.

The experiment was conducted to measure the average response time and the number of aborts for a group of transactions running on a multiversion database system and its corresponding single-version system. Two groups of transactions with different characteristics (e.g., type and number of access to data objects) were executed concurrently. The objective was to study the sensitivity of key parameters on those two performance measures. The details of this study are given in [8]. Here we present our findings briefly.

Performance is highly dependent on the set size of transactions. As shown in Figure 7, a multiversion database system outperforms the corresponding single-version system for the type of workload under which they are expected to be beneficial: a mix of small update transactions and larger read-only transactions. The reason for this is that, in a multiversion database system, a read requests have higher priority than the write requests; whereas the priority for read requests is not provided in a single-version system. Therefore, in a single-version system, the probability of rejecting a read request is equal to that of a write request. The experiment shows that a single-version database system outperforms its multiversion counterpart for a different transaction mix.

It was observed that the performance of a multiversion system in terms of the number of aborts is better than its single-version counterpart for a mix of small update transactions and larger read-only transactions. Similar experiments have been performed by changing the database size and the mean interarrival time of transactions. It was found, however, that the main result remains the same. From these experiments, it becomes clear that among the four variables we studied, the set-size of transactions is the most sensitive parameter for determining the performance of a multiversion database system. This experiment demonstrates the expressive power and performance evaluation capability of the StarLite prototyping environment.

6. Summary

The StarLite system is a prototyping environment that supports the investigation of the properties of application as well as system software: operating systems, database systems, and communication networks. The benefits of real-time software development in a prototyping environment are many-fold: 1) errors are reproducible, 2) design alternatives can be evaluated in a uniform environment, 3) target hardware performance can be scaled, 4) software modules can be dynamically restarted without reloading, and 5) software and experimental results can be easily shared with researchers at different institutions.

The StarLite system supports an "open" systems architecture that allows the application engineer to exercise control over design decisions at every level of the system hierarchy. Not only

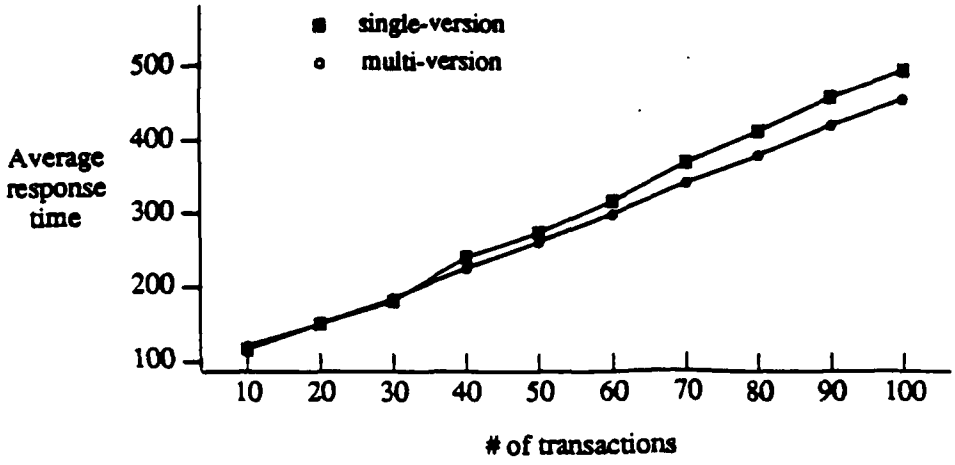
is the system functionally useful at many different interface levels, but also the implementations behind the interfaces can be modified to address application requirements. Furthermore, the system can be easily tailored to different experimental requirements. The implementation is object-oriented in order to support experimentation.

The StarLite system has been operational for a year. It is being used to develop operating systems, distributed database systems, and new network protocols. The architecture has been the "glue" that has enabled the other pieces of the environment to be put together in a way that maximizes a researcher's productivity.

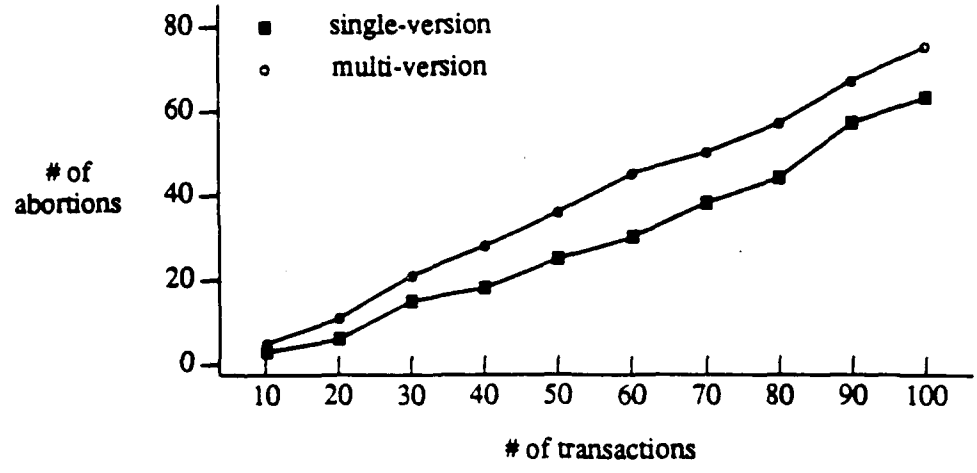
While the initial version of the environment executes as a single UNIX process, future versions could take excellent advantage of both load balancing to distribute a running prototype across a number of machines and of multiprocessor support, such as is found in Mach or Taos.

7. Acknowledgements

The StarLite project is supported by grants of equipment and software from Modula Corporation. The project is funded by the Army Research Office Contract No. DAAL0387-K-090, by the Office of Naval Research Contract No. N00014-86-K-0245, and by IBM Corporation under University Agreement WF-159679.



PARAMETERS
Group 1 : Setsize = 10, Type = READ-only, Transaction Ratio = 80%
Group 2 : Setsize = 2, Type = WRITE-only, Transaction Ratio = 20%



PARAMETERS
Group 1 : Setsize = 2, Type = READ-only, Transaction Ratio = 20%
Group 2 : Setsize = 10, Type = WRITE-only, Transaction Ratio = 80%

Fig. 6. Performance of Multi-version and Single-version Systems

References

(1) Wirth, N., The Personal Computer Lilith, ETH Zurich, Institut fur Informatik Technical Report 40, (April 1981).

(2) Wolfson, O., The Overhead of Locking (and Commit) Protocols in Distributed Databases, *ACM Trans. Database Systems 12*, 3 (Sept. 1987), 453-471.

(3) Canon, M.D. et al, A Virtual Machine Emulator for Performance Evaluation, *Communications of the ACM 23*, 2 (Feb. 1980), 71-80.

(4) Wirth, N., Microprocessor Architectures: A Comparison Based on Code Generation by Compiler, *Communications of the ACM 29*, 10 (Oct. 1986), 978-994.

(5) Rovner, P., Extending Modula-2 To Build Large, Integrated Systems, *IEEE Software 3*, 6 (Nov. 1986), 46-57.

(6) Son, S., A Message-Based Approach to Distributed Database Prototyping, *Fifth IEEE Workshop on Real-Time Software and Operating Systems*, Washington, DC, (May 1988), 71-74.

(7) Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and recovery in Database Systems*, Addison Wesley, 1987.

(8) Son, S. and Y. Kim, A Prototyping Environment for Distributed Database Systems, *Technical Report TR-88-20*, Dept. of Computer Science, University of Virginia, (August 1988).

APPENDIX IV

AN INTRODUCTION TO MODULA-2

AN INTRODUCTION TO MODULA-2

For

Pascal Users

1.0 A Comparison of Pascal and Modula-2

Modula-2[1,2] grew out of a practical need for a general, efficiently implementable, systems programming language. Its ancestors are Pascal[3] and Modula[4]. From the latter, it has inherited the name, the important module concept, and a systematic, modern syntax; from Pascal, most of the rest. This includes in particular the data structures, i.e. arrays, records, variant records, sets, and pointers. Structured statements include the familiar IF, CASE, REPEAT, WHILE, FOR, and WITH statements.

This Report reviews the differences between Pascal and Modula-2. It is not intended to teach you how to program in Modula-2. For that purpose, the definitions by Wirth[1, 2] should be consulted. The implementation assumes that the target computer uses byte-addressing and has a 16-bit word size.

1.1 Identifiers

Identifiers are defined the same as in Pascal. Modula-2, however, is case sensitive. For example, the keyword "IF" is *only* recognized in its all caps form.

Examples:

```
x  scan  starMod  firstLetter  test1
```

1.2 Numbers

The Pascal number format is expanded to allow octal and hexadecimal constants to be expressed. Furthermore, the type CARDINAL is added to explicitly represent unsigned, 16-bit integers, and LONGINT is provided for 32-bit integers. Some important values for these types are as follows:

	MIN( )	MAX( )
INTEGER	-32768	32768
CARDINAL	0	65535
LONGINT	-2147483648	2147483647
REAL	-1.0E-35	1.0E+35

A decimal LONGINT constant is different from an INTEGER in that it must have a "D" following the last digit. Even a very large number must have the D. For an octal or hexadecimal LONGINT, the value of the number must either be too large for an integer or it must have enough leading zeroes to make the number at least six digits long.

Examples:

1980	decimal
3764B	octal (denoted by the trailing "B")
0CADH	hexadecimal (denoted by the trailing "H")
CADH	an identifier, not a number
48H	hexadecimal, leading zero is not required
236713D	decimal LONGINT (denoted by the trailing "D")
356165B	octal LONGINT
000121B	octal LONGINT with leading zeroes
36FA51H	hexadecimal LONGINT
000029H	hexadecimal LONGINT with leading zeroes

REAL numbers are supported in Modula-2 in precisely the same manner as Pascal. All REAL numbers must have a decimal point and must start with a digit; although, digits are not required in the fraction. An exponent field is also supported, but is optional.

Following the fraction portion of the REAL number, an "E" must precede the exponent. The

exponent has a range from -35 to 35. The unary plus ("+") can be placed on positive exponents as an option.

Examples:

5.32	typical REAL
433.	REAL without fraction portion
3.34E-22	REAL with negative exponent
83.28E31	REAL with positive exponent
12.3E+22	also a positive exponent

1.3 Characters and strings

Both the double quote character (") and single quote (') may be used as quote marks. However, the opening and closing marks must be the same character, and this character cannot occur within the string. A string must not extend over the end of a line. A string, consisting of a single-character, is of type CHAR; a string consisting of n>1 characters is of type

ARRAY [0..n-1] OF CHAR.

By convention, many of the library modules use the null character, ASCII code 0, to delimit the end of a string. The storage for constant strings ends with the null automatically. Any string that the user creates should end with the null in order to work properly with string functions.

There is also a notation to represent characters that are not in the language's character set. A sequence of digits terminated with a "C" is interpreted as an octal value of type CHAR. For example, "123C" has the same value and type as "CHAR(123B)".

Examples:

"."	'.'	123C	"Modula"
"Don't Worry!"		'a "quoted" word'	

1.4 Operators, delimiters, and comments

Operators and delimiters are the special characters, character pairs, or reserved words listed below. The reserved words consist exclusively of capital letters and *MUST NOT* be used as identifiers. The Modula-2 symbols, which differ from Pascal's, are listed separately.

Symbols That Are The Same As Pascal

+	-	*	/	:=	.	,
;	(	[	{	}	]	)
^	=	<	>	<>	<=	>=

.	:	AND
ARRAY	BEGIN	CASE
CONST	DIV	DO
ELSE	END	FALSE
FOR	FORWARD	
IF	IN	MOD
NIL	NOT	OF
OR	PROCEDURE	
RECORD	REPEAT	SET
THEN	TO	TRUE
TYPE	UNTIL	VAR
WHILE	WITH	

Symbols Deleted From Pascal

downto	replaced with a BY clause.
file	I/O was deleted from Pascal in favor of services provided by I/O modules.
function	PROCEDURE is used instead.
goto, label	replaced by the LOOP statement.
packed	the only choice in Modula-2.
program	replaced by MODULE.

Symbols In Modula-2 But Not Pascal

|	#	~
BY	DEFINITION	
ELSIF	EXIT	
EXPORT	FROM	
IMPLEMENTATION	IMPORT	
LOOP	MODULE	
POINTER	PROC	
QUALIFIED	RETURN	

Comments may be inserted between any two symbols in a program. A comment is an arbitrary character sequence opened by the bracket "(*" and closed by "*)". Comments may be nested and they

do not affect the meaning of a program. The nesting allows arbitrary sections of a program to be commented out for testing purposes.

1.5 Declarations

As in Pascal, every identifier must be declared within a block. A block in Modula-2, however, can be delimited by either the MODULE or PROCEDURE keyword. Unlike Pascal, the declarations within a block can occur in any order and can be repeated. Another difference is that constant expressions can be used wherever a constant is allowed. Finally, since this implementation is a one-pass compiler, ALL SYMBOLS MUST BE DECLARED BEFORE USE.

1.5.1 Constant declarations

Constant declarations are the same as Pascal, except for the use of constant expressions.

Examples:

```
CONST N    =   100;          (* N stands for 100 *)
  LIMIT    = 2*N-1;          (* LIMIT is for 199 *)
  ODDS     =   BITSET{1, 3, 5}
```

1.5.2 Type declarations

The simple types in Modula-2 consist of enumeration types, subrange types, or type identifiers, which may be qualified. In this context, the term "qualified" means preceded by a module identifier and a period. This option is not present in Pascal. The qualification may be necessary to refer to a type that is in a QUALIFIED EXPORT list or the definition module of another module. The following simple types are denoted by standard identifiers:

INTEGER — A variable of type INTEGER assumes as values the integers between MIN (INTEGER) and MAX (INTEGER).

CARDINAL — A variable of type CARDINAL assumes as values the integers between 0 and MAX (CARDINAL).

BOOLEAN — A variable of this type assumes the truth values TRUE or FALSE. These are the only values of this type, which is predeclared as the enumeration BOOLEAN=(FALSE,TRUE).

CHAR — A variable of this type assumes as values the characters of the ASCII character set.

BITSET — A variable of this type assumes as values any subset of the SET OF [0 .. WordSize-1].

LONGINT — A variable of this types assumes the integer values between MIN (LONGINT) and MAX (LONGINT).

REAL — This type of variable can hold the fractional expressions between MIN (REAL) and MAX (REAL).

PROC — This type is a parameterless procedure.

The type of the bounds for a subrange type, T, is called the base type of the subrange and all operators applicable to operands of type T are also applicable to variables declared with the subrange type name. However, a value to be assigned to a variable of a subrange type must lie within the specified interval. If the lower bound is a non-negative integer, the base type of the subrange is taken to be CARDINAL; if it is a negative integer, it is INTEGER. The only difference from Pascal with respect to enumeration and subrange types is the requirement that a subrange declaration be bracketed.

Examples:

```
TYPE NewInt = INTEGER;
  Color = (RED, BLUE, GREEN);
  Cold  = [-463 .. 58];          (* no brackets in Pascal *)
  Pnew  = POINTER TO ModuleName.New;
                                 (* a qualified reference *)
  Range = [BLUE..GREEN];  (* a subrange of Color *)
  Letter = ["a" .. "z"];         (* the letters "a" to "z" *)
```

Modula-2 handles type equivalence much more

strictly than Pascal. In Pascal, it is perfectly legal to assign variables of two different types as long as the two types "look" alike. Two types look alike if the component parts of the two declarations match exactly. With Modula-2, two separate types cannot be assigned to each other no matter how closely their declarations match.

Example:

```
VAR
    a : ARRAY [0..2] OF INTEGER;
    b : ARRAY [0..2] OF INTEGER;

a := b;  NO!  This is allowed in Pascal, but in
              Modula-2, a and b are variables of
              two different types.
```

1.5.2.1 ARRAY, SET, and POINTER types

The array and pointer types are interpreted and referenced as in Pascal. The array declaration is a bit different in that the bounds list is defined as a list of simple type names, enumerations, or subranges. The pointer declaration is more verbose than in Pascal. The purpose is to make the declaration "stand out" as the "^", used in Pascal, is easily overlooked. As in Pascal, NIL is used to specify an unbound pointer.

One of the exceptions to the "declare before use" rule concerns pointer types. In the case "POINTER TO T", T is automatically treated as a forward reference if it has not already been defined.

Examples:

```
TYPE Demo =
 ARRAY CHAR, (RED, BLUE, GREEN) OF CHAR;
 Array = ARRAY [1 .. 9], [12 .. 347] OF CARDINAL;
 pChar = POINTER TO CHAR;
 pLinks = POINTER TO Links;   (* forward reference *)
 Links = ARRAY [1..4] OF pLinks;   (* defined *)

VAR  x : Demo;  (* referenced with x['j', BLUE] *)
```

Sets are declared as in Pascal but the syntax for a reference to a set constant is different. "{" and "}" are used to bracket set constants, whereas Pascal uses "[" and "]". The element designators can be constants or expressions. Sets are also restricted in size to WordSize elements. This must be a subrange of the integers between 0 and WordSize-1, or a subrange of an enumeration type with at most WordSize values. As a final point, a set constant may be preceded by a type name to document the interpretation of the element list.

Examples:

```
TYPE sColor = SET OF Color;
     BITSET = SET OF [0 .. WordSize-1];
```

Set Constants

{}	the empty set constant
{BLUE, RED}	the union of two colors
sColor{BLUE}	a set consisting of one color
BITSET{0..4, 6}	includes bits 0, 1, 2, 3, 4, 6

1.5.2.2 Record types

The syntax for the Modula-2 record type is similar to the Pascal notation, except for the format of the variant parts. In Pascal, the variant list is parenthesized. In Modula-2, the variant part is implemented as CASE selection. Each sub-declaration (case) in a variant part is delimited by a "|". Also, an ELSE option is provided to denote "all other cases". Another difference is that variant declarations can occur anywhere in a record type declaration, whereas in Pascal, variants are restricted to the end of a record declaration.

Example:

```
TYPE Ex = RECORD
    x,y : BITSET;
    CASE tag0 :Color OF  (* tag0 selects the case *)
        RED, GREEN: a,b : CHAR
      | BLUE:     c: INTEGER
                  (* "|" separates variant parts *)
    END; (* case *)
    z : CARDINAL;
    CASE tag1 : BOOLEAN OF
        TRUE: u,v : INTEGER
      ELSE
          r,s : CARDINAL (* when tag1<>TRUE *)
    END; (* case *)
```

```
END (* Ex *);
```

The example contains two variant sections. The case within the first variant is selected by the value of "tag0", the case within the second variant by "tag1". Remember that, as in Pascal, the variant parts of each case overlay each other in storage.

1.5.2.3 Procedure types

Unlike Pascal, Modula-2 permits variables of procedure type that can have procedure names as values. This feature can be useful when the function to be performed is to be selected at runtime. Since the procedure type is generic, that is, it stands for an arbitrary number of procedure names, the identifiers in the formal parameter list are omitted; only the type names appear. For procedure variables without a formal parameter list, the type PROC may be used.

Examples:

```
TYPE
  prMax  = PROCEDURE(INTEGER, INTEGER)
                                    : INTEGER;
  prSecToDate =PROCEDURE(VAR Seconds) : Date;
  parLess = PROC;
```

Procedure variables are initialized by the assignment of either other procedure variables or procedure constants, which result from procedure declarations.

1.5.3 Variable declarations

Variable declarations serve to introduce variables and associate each with a unique identifier and a fixed data type. Variables whose identifiers appear in the same list all obtain the same type.

Examples:

```
VAR i,j : CARDINAL;
    a   : ARRAY Index OF CHAR;
```

1.5.4 Procedure declarations

Procedure declarations consist of a procedure heading and a block that is called the procedure body. The heading specifies the procedure identifier and the formal parameters. The block contains declarations and statements. The procedure identifier is required at the end of a procedure declaration to document which procedure is being "closed". The primary differences from Pascal are procedure variables, the deletion of the "function" keyword, and the addition of the RETURN statement. Rather than assigning to the procedure identifier to set a return value as in Pascal, a RETURN statement must be used.

```
PROCEDURE identifier [FormalParameters] ";"
  {Const | Type | Var | Procedure | Module Declaration}
[BEGIN
  StatementSequence]
END identifier

FormalParameters =
 "(" [FPSection { ";" FPSection}] ")" [":" qualifiedIdent]

FPSection =
    [VAR] identifierList ":" [ARRAY OF] qualifiedIdent

qualifiedIdent = identifier { "." identifier}
```

The use of a FORWARD qualifier in place of a procedure body allows a procedure to be referenced before its declaration. The FORWARD immediately follows the procedure heading. When the actual procedure is declared, however, the full formal parameter list must be repeated.

Example:

```
PROCEDURE foo (x : CARDINAL);
FORWARD;              (* replaces body *)

PROCEDURE fip;
BEGIN
    foo (14);         (* use before declaration *)
END fip;

PROCEDURE foo (x : CARDINAL);
BEGIN
    InOut.WriteCard (x,4);
END foo;
```

1.5.4.1 Formal parameters

Formal parameters are identifiers that denote actual parameters specified in the procedure call. As in Pascal, both value and variable (VAR) parameters are provided. Formal parameters are local to the procedure, i.e. their scope of reference is the program text that constitutes the procedure declaration.

Example:

```
(* Read a string of digits from the input device. *)
(* The Cardinal value of the digits is returned. *)
(* Conversion starts when a digit is read. *)
(* Conversion stops when a non-digit is read. *)
PROCEDURE ReadCard() : CARDINAL;
    VAR i : CARDINAL; ch : CHAR;
BEGIN
    REPEAT    (* skip characters until a digit is read *)
        InOut.Read(ch);
    UNTIL (ch>="0") AND (ch<="9");
    i := 0;
    REPEAT        (* accumulate the number in "i" *)
        i := 10*i+(ORD(ch)-ORD("0"));
        InOut.Read(ch);
    UNTIL (ch<"0") OR (ch>"9");
    RETURN i;
END ReadCard;
```

The "ReadCard" routine uses the type transfer function, ORD, to manipulate the numeric value of the input character.

Any function with an empty parameter list, such as "ReadCard", must be declared and referenced with the "()" suffix. The goal is to create a visual distinction between a reference to a procedure variable and a procedure call.

The specification of "open" array parameters represents a significant improvement over the static limitations of Pascal. If the parameter is an "open" array, the form

ARRAY OF Type

must be used, where the specification of the actual index bounds is omitted. "Type" must be compatible with the element type of the actual array, and the index ranges are mapped onto the integers 0 to N-1, where N is the number of elements. If the initial array is multidimensional, it is mapped onto the argument with the last subrange listed first. That is if the array's index bounds is defined as [0..2,0..2], the argument will be mapped [0,0]->[0], [0,1]->[1], [0,2]->[2], [1,0]->[3], etc. The "HIGH" standard function can be used to determine "N-1". The example illustrates the use of this feature in an error message routine.

```
PROCEDURE error(VAR message :ARRAY OF CHAR);
        (* Notice: the bound for "message" is omitted *)
    VAR nChar : CARDINAL;
BEGIN
    WriteLn;                        (* skip to new line *)
    FOR nChar := 0 TO HIGH(message)DO
                                    (* no. chars in message *)
        Write(message[nChar]); (* write the message *)
    END; (*for*)
    WriteLn;                        (* skip to new line *)
END error;

error("short");                     error("MEDIUM1");
                error("longest one");
```

The "open" array feature also makes it easy to create libraries of useful routines that can operate over a wide range of input values.

1.5.4.2 Standard procedures

The standard procedures are as follows:

ABS(x)	absolute value; result Type=argType
CAP(ch)	capitalize ch
CHR(x)	the character with ordinal number x
FLOAT(x)	converts x to a REAL value
HIGH(x)	the upper bound of array x
MIN(x)	the minimum value for type x
MAX(x)	the maximum value for type x
ODD(x)	x MOD 2 <> 0
ORD(x)	ordinal number of x in its enumeration
SIZE(x)	the number of words in type x
TRUNC(x)	the LONGINT value of a REAL or the INTEGER value of a LONGINT
LONG(x)	the LONGINT value of an INTEGER or CARDINAL x.

VAL(T, x) is the value with ordinal number x and type T
VAL(T, ORD(x))=x, if x is of type T

```
DEC(x);      x := x-1;
DEC(x, n);   x := x-n;
EXCL(s, i);  s := s-{i}; remove i from set s
HALT;        terminate program execution
INC(x);      x := x+1;
INC(x, n);   x := x+n;
INCL(s, i);  s := s+{i}; include element i in s
```

Examples:

```
ABS(-5) = 5          ODD(3) = TRUE
CHR(65) = 'A'        ORD('A') = 65
CAP('a')= 'A'        VAL(Color, 0) = RED

      x:=8;   y:={0,4,5};

DEC(x); x = 7        DEC(x, 5); x = 3
INC(x); x = 9        INC(x, 5); x = 13
EXCL(y, 4);          y = {0,5}
INCL(y, 6);          y = {0,4,5,6}
```

1.5.4.3 Conversion and Type transfer functions

Conversion functions perform the useful service of converting one number type into another by actually changing the argument's bit values. FLOAT takes an INTEGER, CARDINAL, or LONGINT value and converts it to REAL; FLOAT's inverse, TRUNC, takes a REAL argument and converts it into LONGINT. TRUNC also provides the more docile but no less important role of converting LONGINT values into INTEGER, which involves the removal of the high-order bits.

The other conversion functions perform similar bit additions or removals. LONG takes an INTEGER or CARDINAL value and makes it LONGINT. CHR removes the high byte of an INTEGER or CARDINAL value to make it an ASCII value of type CHAR. ORD, the inverse of CHR, adds a high byte of zeroes back on to create a CARDINAL.

Type transfer functions are different from conversion functions in that they do not change any bits. Type transfer functions merely convert the argument into a new type at compile time. Of course, the new type must have the exact size as the old. ORD, for example, performs a dual role; it is the conversion function mentioned above, and it also gives the ordinal value of its argument in the argument's enumeration. VAL is the inverse of this. It takes the enumeration's type name and its ordinal value and makes them into the enumeration's type. The other way to transfer types is to use the type name as a function. Again, the two types must be of equal size. Type transfer between CARDINAL and INTEGER is automatic on assignment.

Examples:

```
TYPE
        Arr = ARRAY [0..3] OF CARDINAL;
        Rec = RECORD
                m : LONGINT;
                n : LONGINT;
        END;
VAR
        c : CARDINAL;
        i: INTEGER;
        l : LONGINT;
        ch : CHAR;
        r : REAL;
        a : Arr;
        r : Rec;

r := FLOAT(42);          (* r = 42.0 *)
l := TRUNC(r);           (* l = 42D *)
c := TRUNC(l);           (* c = 42  *)
l := LONG(c)             (* l = 42D *)
i := TRUNC(l);           (* i = 42  *)
i := ORD('A');           (* i = 65  *)
ch := CHR(i);            (* ch = 'A' *)
c := 14;                 (* c = 14  *)
i := c;                  (* i = 14  *)
c := i;                  (* c = 14  *)
a := Arr(r);             (* r is made into the array *)
```

1.6 Expressions

The following table defines the interpretation of each operator.

Operator	Meaning
+	integer addition
-	integer subtraction
*	integer multiplication
DIV	integer division
MOD	integer modulus

OR
p OR q means "if p then TRUE, otherwise q"

AND &
p & q means "if p then q, otherwise FALSE"

NOT ~
~ p means "if p then FALSE, otherwise TRUE"

=	compare for equality
<> #	unequal
<	less
<=	less than or equal
>	greater
>=	greater than or equal

IN	contained in, set membership test
+	x IN (s1 + s2) iff (x IN s1) OR (x IN s2)
-	x IN (s1 - s2) iff (x IN s1) & ~ (x IN s2)
*	x IN (s1 * s2) iff (x IN s1) & (x IN s2)
/	x IN (s1 / s2) iff (x IN s1) <> (x IN s2)
<=	p <= q is TRUE if p is a proper subset of q
>=	p >= q is TRUE if q is a proper subset of p

Examples:

```
3+4 = 7                 3-4 = -1
7 DIV 4 = 1             3*4 = 12
7 MOD 4 = 3             TRUE OR FALSE = TRUE
TRUE AND FALSE = FALSE
NOT TRUE = FALSE
3 = 4 is FALSE          3 <> 4 = TRUE
3 < 4 = TRUE            3 <= 4 is TRUE
3 > 4 = FALSE           5 >= 4 is TRUE
5 IN {4,5} = TRUE       {4,5} + {4,7} = {4,5,7}
{4,5} - {4,7} = {5}     {4,5} * {4,7} = {4}
{4,5} / {4,7} = {5,7}   {4,5} <= {4,5,7} = TRUE
{4,5,7} >= {4,5} = TRUE
```

1.7 Statements

The major difference in statement structure from Pascal involves the elimination of the distinction between simple and compound statements. In other words, "BEGIN S {; S} END" has been deleted by making every structured statement a compound statement. REPEAT, for example, was already in this form and required no change. The advantage of the new format is that statements can be arbitrarily added without worrying about whether a "BEGIN-END" is necessary. To facilitate this property, we recommend that every statement be terminated with a semicolon. Except for the compound statement convention, the following statements are similar to the syntax used in Pascal. The WITH statement is restricted to a single record selector.

```
ForStatement =

    FOR identifier ":=" expression TO expression
                [BY ConstExpression] DO
        StatementSequence
    END (* FOR *)

RepeatStatement =

    REPEAT
        StatementSequence
    UNTIL expression

WhileStatement =

    WHILE expression DO
        StatementSequence
    END (* WHILE *)

WithStatement =

    WITH recordReference DO
        StatementSequence
    END (* WITH *)
```

The Modula-2, FOR loop uses the optional BY clause to specify the step value to be used in each iteration. The step must be a constant. If the step is positive, the loop counts up to the TO value. If the step is negative, the loop counts down to the TO

value.

Examples:

```
FOR i := 3 TO 7 DO          i=3,4,5,6,7
FOR i := 3 TO 7 BY 2 DO     i=3,5,7
FOR i := 7 TO 1 BY -2 DO    i=7,5,3,1
```

1.7.1 Assignments and type compatibility

The assignment serves to replace the current value of a variable by a new value indicated by an expression. The assignment operator is written ":=" and is pronounced as *becomes*.

```
assignment =
    variableReference ":=" expression
```

The type of the variable must be assignment compatible with the type of the expression. Operands are said to be assignment compatible, if either they are compatible, or both are of type INTEGER or CARDINAL or subranges with base types INTEGER or CARDINAL. Two operands of types T0 and T1 are compatible if either T1 = T0, or T1 is a subrange of T0, or T0 is a subrange of T1, or if T0 and T1 are overlapping subranges of the same base type. In the case of overlapping subranges, runtime checks for range violations may be necessary to detect errors.

1.7.2 CASE statement

The CASE statement in Modula-2 is somewhat different than the Pascal version. First, subrange constants are allowed as a shorthand notation for a range of case labels.

Pascal	Modula-2
3,4,5,6,7 :	3..7 :

The subrange notation saves typing. Furthermore, constant expressions can also be used as case labels. Thus, defined constants can be used to parameterize selection. Finally, the "|" is used to separate cases and an ELSE clause is adopted as a shorthand for the label standing for *all other labels*. No value may occur more than once as a case label. The maximum number of cases per case statement is 256.

```
CaseStatement =
        CASE expression OF
        Case
        {"|"Case}
        [ELSE
                StatementSequence]
        END (* CASE *)

Case =
        [CaseLabels {"," CaseLabels} ":"
                StatementSequence]

CaseLabels =
        ConstExpression [".." ConstExpression]
```

Example:

```
(* Read a string of digits from the input device. *)
(* "," and "." are allowed in the string for readability. *)
(* The Cardinal value of the digits is returned. *)
(* Conversion starts when a digit is read. *)
(* Conversion stops when a non-digit is read. *)
PROCEDURE ReadCard() : CARDINAL;
    VAR i : CARDINAL;
        ch : CHAR;
BEGIN
    REPEAT    (* skip characters until a digit is read *)
        InOut.Read(ch);
    UNTIL (ch>="0") AND (ch<="9");
    i := 0;
    LOOP            (* accumulate the number in "i" *)
        CASE ch OF
        "0".."9":  i := 10*i+(ORD(ch)-ORD("0"));
        | ",", ".":                 (* ignore "," and "." *)
        ELSE                        (* stop at non-digit *)
            EXIT; (* loop *)
        END; (* case *)
        InOut.Read(ch);
    END; (* loop *)
    RETURN i;
END ReadCard;
```

1.7.3 IF statement

The IF statement has been modified by the addition of an ELSIF clause whose purpose is to provide a shorthand notation for tests that, in Pascal, would require multiple IF statements.

```
IfStatement =
```

```
IF expression THEN
        StatementSequence
{ELSIF expression THEN (*zero or more *)
        StatementSequence}
[ELSE                   (*zero or one ELSE *)
        StatementSequence]
END (* IF *)
```

Example:

Pascal	*Modula-2*
if x = 1 then	IF x = 1 THEN
y := 2	y := 2;
else if x = 9 then	ELSIF x = 9 THEN
y := 3	y := 3;
else	ELSE
y := 6;	y := 6;
	END; (* IF *)

The expressions following the symbols IF and ELSIF are of type BOOLEAN. They are evaluated in the sequence of their occurrence until one yields the value TRUE. Then, the associated statement sequence is executed and the IF terminated. If an ELSE clause is present, it is executed if and only if all Boolean expressions yielded the value FALSE, much like the ELSE in the CASE construct.

1.7.4 LOOP and EXIT statements

A loop statement specifies the continuous execution of a statement sequence. This statement is used quite frequently in concurrent algorithms because, unlike sequential programs, termination is often undesirable. Imagine what would happen if an operating system halted after 10,000 iterations.

```
LoopStatement =
        LOOP
            StatementSequence
        END (* LOOP *)

ExitStatement = EXIT
```

Example:

```
TYPE pList = POINTER TO List;
    List = RECORD
        link : pList;           (* a singly-linked list *)
        attribute : Attribute;  (* a list element      *)
    END; (* List *)

PROCEDURE search(list : pList;
                 VAR attribute : Attribute) : BOOLEAN;
    (* check to see if "attribute" is in "list" *)
BEGIN
    LOOP (* search singly-linked list *)
        IF list = NIL THEN
            EXIT;
        ELSIF attribute = list^.attribute THEN
            RETURN TRUE; (* attribute is in the list *)
        END; (* IF *)
        list := list^.link;    (* advance to next element *)
    END; (* LOOP *)
    RETURN FALSE;          (* end of list; not there *)
END search;
```

The EXIT statement specifies termination of the loop and, when executed, causes execution to continue at the statement following the loop statement. An EXIT statement may terminate a LOOP even if it is nested within other structured statements. Only the closest, enclosing LOOP is terminated.

1.7.5 RETURN statement

The RETURN statement provides a convenient way to leave a procedure as soon as an exit condition becomes true. In Pascal, a procedure can only be terminated by executing the "end" of the block, which is often an inconvenience.

RETURN [expression]

In Modula-2, the RETURN statement serves the dual role of specifying the result for a function and of returning to the caller for both subroutines and functions. For a subroutine, the expression must be omitted. For a function, it must be present. The expression, representing the returned value, must match the type specified for a function.

2.0 Programming Conventions

In addition to the indentation conventions used in the Modula-2 definition, you should try to, and we will, adhere to the following programming conventions. Hopefully, the result will be visually pleasing programs that are easier to understand due

to the presence of syntactic cues.

2.1 Names and declarations

Declarations should help document the use of a variable; thus, try to use subrange and enumerated type declarations instead of INTEGER. Most identifiers should be written in lower case, except for the first letter of each new *word*, that should be capitalized.

```
line   firstLine   nextLineOffset
```

Capitalize the first letter of type identifiers, module names, and the names of exported procedures; capitalize all letters of CONST definitions. If the name of a constant is several words, just capitalize the first two letters of the first word(e.g. CHarsPerWord). Try to use full words for all names. However, if space is a problem, the following shorthand conventions can be used.

Choose a short *tag* for each basic type that you create, e.g. Ln for Line or Buf for Buffer. Use the following prefixes to construct tags for derived types:

```
p - pointer to:     pBuf = POINTER TO Buf
i - index for:      iLn  = index for ARRAY OF Ln
s - set of:         sColor=SET OF Color
sr- subrange of:    srColor=[BLUE..GREEN]
n - length of:      nString=number of characters in
```

If you need only one variable of a given type in a scope, use the tag as its name:

```
buf : Buf
```

If you need several names, append modifiers (avoid simple numbers like 1, 2, etc.):

```
bufOld, bufNew, bufAlt : Buf
```

2.2 Layout

Try to follow the indentation examples in the Modula-2 definition. Write one statement per line, unless several simple statements, *which together perform a single function*, will fit on one line. It is acceptable to put a loop on a single line if it will fit. If a statement will not fit on a single line, indent the continuation line(s).

A semicolon follows the last statement in a statement sequence and the last field in a field list. The purpose is to make insertions and deletions less error-prone.

Each DEFINITION module should be commented to describe its general function. Also, each exported procedure should have a brief comment. In addition, it is advisable to comment VAR parameters as "IN", "OUT", or "INOUT" to denote the presence or absence of side-effects.

2.3 Spaces

Leave a space after a comma or semicolon and none before; leave a space before and after a colon. Surround ":=" with spaces. A space should appear after left-comment and before right comment. Don't put spaces inside brackets or parentheses or around single-character operations.

3.0 Changes to Modula-2

The following list reflects a number of changes to the Modula-2 definition[5]. The changes resulted from a meeting between Wirth and representatives of several firms that had implemented Modula-2.

1. All objects declared in a definition module are exported. The explicit export list is discarded. The definition module may be regarded as the implementation module's separated and extended export list.

```
DEFINITION MODULE identifier ";"
   {import}
   {definition}
END identifier "."
```

2. The syntax of a variant record type declaration is changed so that the ":" is always required. The presence of the colon makes it evident which part was omitted, if any.

```
CASE [identifier] ":" qualifiedIdent OF
```

3. The syntax of the case statement and the variant

record declaration is changed so that either may be empty. The inclusion of the empty case and empty variant allows the insertion of superfluous bars similar to the insertion of superfluous semicolons for empty statements.

4. A string consisting of N characters is said to have length N. A string of length 1 is compatible with the type CHAR.

5. The syntax of the subrange type is changed to allow the specification of an identifier designating the base type of the subrange. Example: INTEGER[0 .. 99].

6. The syntax of sets is changed to allow expressions as set element selectors.

```
set = [qualifiedIdent] "{" [element {"," element}] "}"
element = expression [".." expression]
```

7. The character "~" is a synonym for NOT.

8. The identifiers LONGCARD, LONGINT, and LONGREAL denote standard types (which may not be available on some implementations).

9. The type ADDRESS is compatible with all pointer types and with either LONGCARD or LONGINT depending on the implementation.

10. The new standard functions MIN and MAX take as an argument any scalar type, including REAL. They stand for the type's minimal/maximal value.

REFERENCES

[1] Wirth, N., Modula-2. Technical Report No. 36, Institut fur Informatik der ETH Zurich, (Dec. 1980).

[2] Wirth, N., Programming in Modula-2. Springer-Verlag New York Inc., (1982).

[3] Wirth, N. and K. Jensen., Pascal user manual and report. Springer-Verlag New York Inc., (1976).

[4] Wirth, N., Modula: a language for modular multiprogramming. Software—Practice and E. ... nce 7, (1977), 3-35.

[5] Wirth, N., Schemes for multiprogramming and their implementation in Modula-2. Technical Report No. 59, Institut fur Informatik der ETH Zurich, (June 1984).

APPENDIX V

AN INTRODUCTION TO MODULAR PROGRAMMING

AN INTRODUCTION TO MODULAR PROGRAMMING

1.0 Introduction

Modula-2 was designed to support modular programming. This section outlines the features of Modula-2 which reflect that goal. Also, the facilities of Modula-2 for systems programming are illustrated.

Many systems today are large programs, ranging in size from ten thousand to one-half million lines of code. Obviously, some design guidelines are necessary to manage the complexity of implementing and maintaining such large systems. The most successful approach has been to use modular programming techniques[1] that allow one module to be written with little knowledge of the implementation of other modules and that allow modules to be recompiled and replaced without requiring recompilation of an entire system. The expected benefits of modular programming are shortened development time for new products because modules can be implemented by separate groups, increased flexibility because the implementation of one module can be changed without the need to change others, and increased comprehensibility because the system can be studied one module at a time.

In system design, the first step is to partition the specification into a number of modules with well-defined interfaces. At this point, only the interfaces are considered, not the module implementations. Each module should be small and simple enough to be thoroughly understood and well programmed. The intention is to describe all "system level" decisions (i.e. decisions that affect more than one module). The modularization must take into account both the functions to be provided to users, resulting in top-down decisions, and the technological constraints imposed by the possible execution environments, resulting in bottom-up decisions.

In choosing a modularization for a system, it is advantageous to impose a hierarchical organization on the modules. A hierarchical structure results when all modules at level *i* in a system use only modules at levels lower than *i* for their implementation. A module at level 0 is implemented without referring to any other modules. The existence of a hierarchical structure assures us that upper levels can be deleted and arbitrarily rebuilt. This property enhances the extensibility, or "open"ess, of a system. If "low-level" modules were implemented such that they depended upon "high-level" modules, a hierarchy would not exist and it would be much more difficult to delete or update portions of the system.

2.0 Modular Programming

The following table illustrates the syntax of a compilation unit in Modula-2.

Modula-2 Program Structure

```
CompilationUnit = DefinitionModule |
        [IMPLEMENTATION] ProgramModule

ProgramModule =
        MODULE identifier";"
            {import}
        block identifier "."

DefinitionModule =
        DEFINITION MODULE identifier ";"
            {import}
```

```
        {definition}
      END identifier "."

import =
      [FROM identifier] IMPORT IdentifierList ";"

export =
     EXPORT [QUALIFIED] IdentifierList ";"

definition =
      CONST {ConstantDeclaration ";"}  |
      TYPE  {identifier ["=" type] ";"} |
      VAR   {VariableDeclaration ";"}  |
      PROCEDURE identifier [FormalParameters] ";"
```

A program module encapsulates the implementation of an abstraction. A compiler, for example, might have modules for symbol table lookup, reading from the input stream, accumulating tokens, and generating code.

To meet our modularity requirements, a module must be easily recognized. In addition, its function should be easy to determine. This does not mean examining the listing of the entire module. In fact, for proprietary software, the module listing may not be available. As you will learn in this section, Modula-2 meets, and exceeds, all of our requirements. We start with an example of a Modula-2 program module that prints the integers between one and a hundred, and their squares.

```
PRINT THE SQUARES OF THE INTEGERS 1..100

MODULE Main;

   FROM InOut IMPORT       (* Procedures *)
      WriteCard, WriteString, WriteLn;
   VAR i : [1..100];

BEGIN
   WriteString("Number  Number Squared");
   WriteLn;
   FOR i := 1 TO 100 DO
      WriteCard(i, 4);  (* aligns number under "b" *)
      WriteCard(i*i, 16);       (* aligns under "S" *)
      WriteLn;                  (* writes end-of-line  *)
   END; (* for *)
END Main.
```

The major difference between the Modula-2 version of the program and its Pascal equivalent is the IMPORT list and the variety of I/O procedures. Modula-2 has no builtin I/O statements; therefore, all I/O is performed with procedures written in Modula-2. This design decision resulted in a simpler implementation for the compiler but increased typing for users.

The IMPORT list is necessary to tell the compiler where to find the definitions for the "Write" procedures, in this case in module "InOut", which has been separately compiled. The IMPORT list also enumerates the symbols from "InOut" that are required by the "Main" program.

"InOut" is an example of a low-level module that can be used over and over again by high-level modules. In fact, program modules, such as "Main", must always occur at the highest system level as they can only "import" definitions from lower-level modules like "InOut".

If a global variable is not listed in the IMPORT list, it is invisible to the module. Thus, by examining the interface specification at the top of a program module, a user can determine what services the module depends on from its environment (useful documentation). Since the modularization process starts by defining module interfaces, the IMPORT list is usually determined prior to implementation. Any symbol that is used by a module and does not appear in the IMPORT list *must* be declared in the body of the module.

If the FROM clause in an IMPORT list is omitted, the list of identifiers must name modules, not symbols contained in modules. In this case, all of the symbols occurring in the DEFINITION part of the named modules are made available to the program. However, these symbols can only be referenced via a qualified name of the form ModuleId.SymbolId. The following example

illustrates the qualified name option.

```
PRINT THE SQUARES OF THE INTEGERS 1..100

MODULE Main;

  IMPORT InOut;    (* only the module name *)
  VAR i : [1..100];

BEGIN
  InOut.WriteString("Number  Number Squared");
  InOut.WriteLn;
  FOR i := 1 TO 100 DO
    InOut.WriteCard(i, 4); (* aligns number under "b" *)
    InOut.WriteCard(i*i, 16);    (* aligns under "S"*)
    InOut.WriteLn;           (* writes end-of-line    *)
  END; (* for *)
END Main.
```

2.1 DEFINITION modules

Modula-2 permits the *definition* specification for a module to be separated from the module's *implementation*. The two parts can be compiled separately but must, of course, match with respect to declarations. A DEFINITION module supports information hiding by eliminating the implementation code. It is intended to be standalone documentation for the users of an abstraction. Furthermore, in most Modula-2 implementations, the IMPLEMENTATION part can be recompiled arbitrarily without causing additional recompilations on the part of its users. If a DEFINITION module is recompiled, all modules that refer to it must be recompiled.

A DEFINITION module contains only the constant, type, variable, and procedure-heading declarations that are necessary to use the corresponding IMPLEMENTATION module. The interface specification lists the entities that are "export"ed to the outside world by the module and any entities from the outside world that are "import"ed (used) by the DEFINITION module. The following example illustrates a portion of the "InOut" DEFINITION module. Notice that only the procedure headings are given. The procedure bodies are specified in the IMPLEMENTATION module for "InOut".

THE InOut DEFINITION MODULE

```
(* Provides formatted I/O services for basic types *)
DEFINITION MODULE InOut;

PROCEDURE WriteCard(x, n : CARDINAL);
(* write cardinal x with (at least) n characters.
   If n is greater than the number of digits needed,
   blanks are added preceding the number. *)
PROCEDURE WriteLn(); (* terminate the current line *)
PROCEDURE Write(ch:CHAR);
(* write a single character *)
PROCEDURE WriteString(s : ARRAY OF CHAR);
(* write HIGH(s)+1 characters from s *)

END InOut.
```

The full details of types exported from DEFINITION modules are visible to importing modules. If an enumeration or record type is exported, the enumerated constant and field names are automatically exported as well. This is termed a *transparent* export.

At the other extreme, it is possible to export only a type's name. This is referred to as *opaque* export. The term "opaque" denotes the hiding of the details of a type's implementation from its users. An opaque type is declared as follows:

An Opaque Type Declaration

```
TYPE identifier;
```

In the corresponding IMPLEMENTATION module, an opaque type can only be declared as a pointer or a simple type, such as CARDINAL. Instances of opaque types can be used only for assignment, comparison, or as arguments to procedures defined in the corresponding IMPLEMENTATION module.

2.2 IMPLEMENTATION modules

A correctly structured module has the property that its implementation can be changed without changing the parts of the program outside the module. This property by itself would

suffice as a reason to use Modula-2 over Pascal.

It is important to document the external symbols that are used in an IMPLEMENTATION module. Notice that the IMPORT list for the DEFINITION and IMPLEMENTATION parts need not match. Typically, the IMPLEMENTATION module's list will be longer as greater detail is necessary to implement an abstraction as opposed to specifying it.

Every IMPLEMENTATION module contains an initialization part, following the "BEGIN", that is used to put the module into a consistent state before program execution starts. The initialization code is executed by the runtime system before the main program begins. Therefore, it is unwise to put infinite loops in an initialization part.

The next example illustrates the use of a DEFINITION and IMPLEMENTATION module to define a stack manipulation utility. The program implements a single stack that has its size and its element's type chosen by its users. In the example, "stack" and "iStack" are not exported because they are implementation details. By "hiding" them, the programmer responsible for maintaining the module can continue to refine and improve its implementation without affecting any of its users. For instance, the stack could be implemented as a linked list rather than an array.

In addition to serving as a convenient organizational tool, the module also provides an information-hiding and parameterization service. The user of the module can call "Push", "SetEmpty" and "Pop", but all implementation details are hidden. In the example, the module imports the type of the stack's elements and the size of the stack. Thus, this module could be used to create the following varieties of stacks.

Possible Content of the "Parameters" Module

```
CONST MAxStackSize = 42;
TYPE StackType = INTEGER; (*a stack of 42 integers*)
```

```
CONST MAxStackSize = 97;
TYPE StackType =BOOLEAN;
        (*a stack of 97 Booleans*)
```

The advantage of this parameterization is that the stack module takes on a life of its own, independent of any particular program. Any algorithm that needs a stack can "check out" this module from a system library, read its specification, set up the parameters, and not worry about coding it. Notice that, unlike procedure parameters, the imported type and constant are evaluated and have their effect only at compile time.

A Stack Manipulation Example

```
(* This module implements a single stack together with the
   operators that manipulate it. To use this module, create
   a Parameters module that defines MAxStackSize, which
   is the number of elements desired, and StackType. *)
DEFINITION MODULE StackManipulation;

FROM Parameters IMPORT
    (*Type*) StackType; (* restricted to a simple type *)

PROCEDURE Push(stackElement : StackType)
                                        :BOOLEAN;
(* adds to top; returns FALSE if a push doesn't succeed *)
PROCEDURE Pop(VAR stackElement : StackType)
                                        :BOOLEAN;
(* removes from top; returns FALSE if stack was empty *)
PROCEDURE SetEmpty();
(* sets the stack to empty *)

END StackManipulation.


IMPLEMENTATION MODULE StackManipulation;

(* ********INTERFACE SPECIFICATION******** *)
FROM Parameters IMPORT
    (*Const*) MAxStackSize, (*Type*) StackType;
(* ************DECLARATIONS************** *)
VAR
    stack : ARRAY [1 .. MAxStackSize] OF StackType;
    iStack : [1 .. MAxStackSize+1];

(* **********IMPLEMENTATION PART******** *)
PROCEDURE Push(stackElement : StackType)
                                        :BOOLEAN;
BEGIN
```

```
    IF iStack <= MAxStackSize THEN
        stack[iStack] := stackElement;
        INC(iStack);  (* the same as iStack:=iStack+1 *)
        RETURN TRUE;
    ELSE
        RETURN FALSE;      (* error-stack overflow *)
    END; (* if *)
END Push;

PROCEDURE Pop(VAR stackElement : StackType)
                                        :BOOLEAN;
BEGIN
    IF iStack > 1 THEN
        DEC(iStack);   (* the same as iStack:=iStack-1 *)
        stackElement := stack[iStack];
                        (* exit with a value *)
        RETURN TRUE;
    ELSE
        RETURN FALSE;      (* error-stack underflow *)
    END; (* if *)
END Pop;

PROCEDURE SetEmpty();
BEGIN
    iStack := 1;
END SetEmpty;

(* **********INITIALIZATION PART********** *)
BEGIN
    SetEmpty();
END StackManipulation.
```

2.3 Module-based abstractions

In this Section, we review some of the more common techniques for implementing a data abstraction. System designers must choose among these methods when designing the user interfaces. The previous StackManipulation example illustrates one of the choices. Notice that it is restricted to implementing exactly one stack per use of the module. The other data abstraction choices are to export a type, to export an opaque type, and to export an index. The StackManipulation module is used as an example for each method.

2.3.1 Exported type

The first choice to implement an abstraction is to export a type, such as "StackOfIntegers". The advantage of this approach is that the new abstraction extends the language available to the programmer. The new type can be used to declare variables in the same way as any builtin type like INTEGER or CHAR. Instances of these variables are then passed as arguments to the StackManipulation procedures.

The disadvantage of the approach is that the implementation details of the type are visible and accessible to the users. As a result, a change in representation requires a recompilation by all users of the module and may invalidate some programs. Thus, this design choice should be used with extreme care for any user interface provided by an operating system. Another disadvantage is the inability to share at runtime a single StackManipulation module for stacks of different type.

A Stack Manipulation Example With An Exported Type

```
DEFINITION MODULE StackManipulation;

FROM Parameters IMPORT
    (* Const *) MAxStackSize,
                        (*number of stack elements*)
    (*Type*) StackType;         (* the element type *)

(* This module implements a stack type together with the
   operators that manipulate it. To use this module, create
   a Parameters module that defines MAxStackSize, which
   is the number of elements desired, and StackType, which
   can be of any type. *)
TYPE Stack = RECORD
    iStack: [1..MAxStackSize+1];
    stack : ARRAY [1..MAxStackSize] OF StackType;
END; (*Stack*)

PROCEDURE Push(VAR stack : Stack; VAR element :
                                    StackType):BOOLEAN;
(* adds to top; returns FALSE if a push doesn't succeed *)
PROCEDURE Pop(VAR stack : Stack; VAR element :
                                    StackType):BOOLEAN;
(* pops from top to "element";
  returns FALSE if stack was empty *)
PROCEDURE SetEmpty(VAR stack : Stack);
(* sets a stack to empty *)

END StackManipulation.
```

2.3.2 Opaque type

The second technique uses an opaque type, a pointer, to represent the stack abstraction. When the user declares instances of the Stack type, only uninitialized pointers are allocated. Thus, the implementation must provide a "NewStack" operator to allocate a stack of a particular size and a "FreeStack" operator to deallocate stacks.

A Stack Manipulation Example With An Opaque Type

```
DEFINITION:

TYPE Stack;
PROCEDURE NewStack(VAR stack : Stack;
                   stackSize : CARDINAL) :BOOLEAN;
(* allocate stack;
  return FALSE on storage allocation error *)
PROCEDURE FreeStack(VAR stack:Stack):BOOLEAN;
(* deallocate stack;
   return FALSE on storage allocation error *)

IMPLEMENTATION:

TYPE Stack = POINTER TO StackDescriptor;
StackDescriptor = RECORD
     allocated: BOOLEAN;(* set to TRUE by NewStack *)
     size : CARDINAL;          (* set from stackSize *)
     iStack : [1..MAxStackSize+1];
     pStack : POINTER TO ARRAY [1..MAxStackSize]
                               OF StackType;
END; (* StackDescriptor *)

PROCEDURE NewStack(VAR stack : Stack; stackSize :
                        CARDINAL) : BOOLEAN;

BEGIN
  IF (stackSize=0) OR (stackSize>MAxStackSize) THEN
    RETURN FALSE;
  END;
  Storage.ALLOCATE(stack, TSIZE(StackDescriptor));
  IF stack = NIL THEN
    RETURN FALSE;
  END;
  stack^.allocated := TRUE;
  stack^.size := stackSize;
  stack^.iStack := 1;
  Storage.ALLOCATE(stack^.pStack, stackSize);
  IF stack^.pStack = NIL THEN
    Storage.DEALLOCATE(stack,
                       TSIZE (StackDescriptor));
    RETURN FALSE;
  END; (* if *)
  RETURN TRUE;
END NewStack;
```

The advantage of this approach is the ability to bind the size of a stack at runtime. In other words, the IMPLEMENTATION module must allocate the space for each new stack. The disadvantage is again the inability to define a "class" of stacks that would allow the component type to be specified arbitrarily.

2.3.3 Index

The last option uses the same DEFINITION module as the previous example. But in this case, the opaque type is declared as a CARDINAL rather than a pointer. The IMPLEMENTATION module maintains an array of pointers to StackDescriptors. The array index, which is used as the argument to the module's procedures, selects a descriptor from the array. The pointer from the descriptor is then used to manipulate a stack, just as was done with the previous example. The array simply represents an additional level of indirection. The advantage of the index technique is that it supports validity checking. That is, it is easy to determine if a given index is really associated with a stack. Validity checking is more difficult when using pointers since there is no way to force a user to initialize instances of the "Stack" type.

3.0 Low-Level Programming Facilities

In order to implement some systems in Modula-2, it must be possible to deal with machine dependencies and it must be possible to bypass the compiler's type checking. We discuss the latter requirement first. (These low-level operations should be used carefully and only when *absolutely* necessary.

3.1 Eliminating type checking

The first facility to breach Modula-2's type

checking is type transfer functions. A type identifier can be used as a function to transfer a parameter to the type identifier's type. In most implementations, no conversion is performed; type transfers have their effect at compile time.

Type Transfer Examples

```
CHAR(65)      =  'A'
CARDINAL('A') =   65
BITSET(3)+BITSET(5) = 7
```

3.2 The SYSTEM module

The second set of capabilities is provided by module SYSTEM, which is "builtin" to the compiler. The definition of SYSTEM is implementation dependent.

Low-Level SYSTEM Facilities

```
DEFINITION MODULE SYSTEM;
(* IMPLEMENTATION DEPENDENT *)

TYPE
    ADDRESS=POINTER TO WORD;
    (*assignment compatible with pointer types*)
    WORD;        (* compatible with any simple type *)
PROCEDURE ADR(x : (***ANY TYPE***) )
                                    : ADDRESS;
(* turns any variable reference into an ADDRESS type. *)
PROCEDURE TSIZE(x : (*ANY TYPE IDENTIFIER*) )
                                    : CARDINAL;
(* returns the number of address units that "x" occupies.
It operates on a type's name, not on instances of the type. *)

END SYSTEM.
```

The SIZE (builtin) and TSIZE functions allow the implementor to obtain machine specific information. For example, the size of an integer array big enough to store a 512-word disk sector can be obtained with the expression "512 DIV TSIZE(INTEGER)". Since the size of a word in our implementation is one machine unit, TSIZE(INTEGER) returns the value one. The use of these functions improves the portability of an operating system.

The ADDRESS and WORD types support the implementation of generic routines, particularly for I/O. Both types bypass the compiler's type checking. Modula-2 also supports the convention that if a formal parameter is specified as ARRAY OF WORD, then any variable, structured or unstructured, can be supplied as an argument. The ADR function can be used to initialize a pointer to the address of any data structure. As an example, the following routine takes an arbitrary array of characters and prints it in slices of "unit" characters at a time.

Print Slices of Strings

```
PROCEDURE printSlice(VAR s:ARRAY OF WORD;
                              size, width:CARDINAL);
VAR
    i,j:CARDINAL;
    c:POINTER TO ARRAY [0..9999] OF CHAR;
BEGIN
    j := 0;
    c := ADR(s);              (* use a pointer to access *)
    FOR i := 0 TO size-1 DO (* byte-wise for each    *)
        InOut.Write(c^[i]);    (* char in the argument *)
        INC(j);
        IF j >= width THEN
                          (* print "width" characters *)
            InOut.WriteLn;  (* then start a new line  *)
            j := 0;
        END; (* if *)
    END; (* for *)
    IF j <> 0 THEN
        InOut.WriteLn;
    END;   (* end line, if necessary *)
END printSlice;
```

Examples:

```
a := '0123456789';
printSlice(a, 10, 5);     prints     01234   56789
printSlice(a, 10, 3);     prints     012  345  678  9
```

3.3 Coroutines

The final low-level facility that is discussed is the notion of a *coroutine*. Wirth uses this abstraction to build higher-level operating system rounnes to manipulate a program; for example, to assign a

program the CPU or to remove it from control of the CPU. The coroutine operators are fundamental to any operating system.

In a subroutine program structure, there is a master/slave relationship between a calling program and its subroutine. Usually, a subroutine has one entry point and all local variables, except the formal parameters, are undefined at entry time.

Coroutines, on the other hand, are programs that may call each other, but do not have a master/slave relationship. On exit from a coroutine, its state (i.e. program counter, stack pointer) is saved in a variable of type Coroutine; the next time the coroutine is called, it *resumes* execution at exactly the point where it previously paused. All local variables and parameters retain their previous values.

The Coroutine type and the operators to manipulate coroutines are defined in the COROUTINE module, which again is machine dependent.

In Modula-2, a coroutine is created by specifying a procedure, which represents the actions of the coroutine, and a stack to hold the procedure activation records, which represent the execution state of the procedure. Before a coroutine can be "resumed" for the first time (e.g. start execution), its state must be initialized by calling InitCoroutine. The arguments to InitCoroutine are a procedure as well as a stack base address and size. The stack size must be chosen in an application-dependent way; in fact, some architectures do not even require this information.

The Transfer procedure implements the "resume" operation by saving the execution state of the current coroutine in a variable of type Coroutine and restoring the execution state of a second coroutine. A RETURN operation from a coroutine procedure is normally an error.

The COROUTINE Module

```
DEFINITION MODULE COROUTINE;
(* Routines to turn procedures into coroutines and to
   transfer control of the CPU from one coroutine to
   another. *)

TYPE
   Coroutine = POINTER TO RECORD
                    (* stores state of a coroutine *)
      pc : ADDRESS;
         (*bare machine's program counter *)
      sp : ADDRESS;
         (*bare machine's stack pointer *)
   (* ANY OTHER DATA NEEDED TO
     EXECUTE A Coroutine*)
   END RECORD;

PROCEDURE InitCoroutine(p:PROC; stack:ADDRESS;
        stackSize:CARDINAL;
                VAR (* OUT *)coroutine:Coroutine);
(* Initializes a coroutine record for procedure "p" so
   that a "Transfer" to "p" will start it executing. *)
PROCEDURE Transfer(VAR from, to : Coroutine);
(* Saves the hardware registers of the executing
   procedure in "from" and then resets the registers to
   the values in "to", resulting in a transfer of control. *)

END COROUTINE.
```

The following example uses three coroutines to illustrate the concepts. The first coroutine, "getChar", is used as a filter to reduce all sequences of three identical characters to the letter "J". Thus, "abbbbabbddd" as input would result in "aJbabbJ" as output. The second coroutine, "print", "resumes" the first to retrieve and print filtered characters. Since the "Main" program is initialized with a stack, it is automatically a coroutine.

When the "getChar" routine pauses, it leaves the filtered character in "resultChar". The program stops when it reads a ".", followed by any different character. Notice that the values of "ch" and "previousChar" in "getChar" are saved across Transfer operations.

A Coroutine Example

```
MODULE Main;
IMPORT InOut, COROUTINE;
VAR
   startCo, getCo, printCo : COROUTINE.Coroutine;
   stack1, stack2 : ARRAY [1..200] OF INTEGER;
         (* stack space *)
   resultChar : CHAR;
```

```
PROCEDURE print();
BEGIN
    REPEAT
        COROUTINE.Transfer(printCo, getCo);
            (* resume "getChar" coroutine *)
        InOut.Write(resultChar);
    UNTIL resultChar = ".";   (* stop on "." sequence *)
    COROUTINE.Transfer(printCo, startCo);
            (* resume "main" program *)
END print;

PROCEDURE getChar();
VAR ch, previousChar : CHAR;
        (* these values are preserved *)
BEGIN
    InOut.Read(previousChar);
    LOOP
        InOut.Read(resultChar);
        IF previousChar = resultChar THEN
                    (* do two in a row match? *)
            InOut.Read(resultChar);
            IF previousChar = resultChar THEN
                    (* do three in a row match? *)
                InOut.Read(previousChar);
                resultChar := "J";
            ELSE
                ch := resultChar;
                    (* no, return two, then proceed *)
                resultChar := previousChar;
                COROUTINE.Transfer(getCo, printCo);
                            (* resume "print" *)
                resultChar := previousChar;
                        (* falls through to Transfer *)
                previousChar := ch;
            END; (* if *)
        ELSE
            ch := previousChar;
                    (* two characters are different *)
            previousChar := resultChar;
            resultChar := ch;       (* set return value *)
        END; (* if *)
        COROUTINE.Transfer(getCo, printCo);
                        (* resume "print" coroutine *)
    END; (* loop *)
END getChar;

BEGIN
    COROUTINE.InitCoroutine(getChar, stack1,
                            SIZE(stack1), getCo);
    COROUTINE.InitCoroutine (print, stack2,
                            SIZE(stack2), printCo);
    COROUTINE.Transfer(startCo, printCo);
                    (* save "Main"; resume "print" *)
    InOut.WriteLn;
    InOut.WriteString("End Of Program");
END Main.
```

4.0 Compiling and Executing

The Modula-2 environment is composed of three units: two compilers and one runtime. The compilers perform all the needed code generation, and the runtime executes the code once it is selected.

4.1 The Definition Compiler

The first compiler is called "d" for definition compiler. The definition compiler is the precursor to the second compiler. Its job is to decipher definition modules to produce the implementation interface. Every file "d" receives must have the suffix ".def". If this suffix is not supplied, the compiler will add it automatically. The implementation's interface is stored in a file with the same filename as the source code except the suffix is changed to ".SBL". An implementation interface is required for each reference to an imported module. The compiler searches for any needed .SBL files in the current directory. If it does not find one, it prompts the user to input the path name that locates the needed file.

All definition modules must be compiled before they are used (imported). Once a definition module is compiled, it should not be compiled again unless it is extended. When a definition module is changed, first, compile all dependent definition modules and then secondly, compile all dependent implementation modules.

4.2 The Program Compiler

The second compiler is called "c" for compile. This program takes the ASCII file of a program or implementation module and forms it into the object code that the runtime uses. Files sent to the compiler require the ".mod" suffix in order for the compiler to recognize it as Modula-2 source code. If the suffix is omitted, the compiler will append it

automatically. Also when the suffix is omitted, any error messages generated during compliation will be immediately printed. Object files have the same name as the source text with the suffix switched to ".OBJ".

Error messages generated by either compiler can always be found in the file called filename.LST where filename is the name of the file that the compiler attempted to translate. When errors occur, the compiler will try to continue compilation beyond the error. This is so that all errors can be discovered before the user attempts to compile again. When any errors occur during compilation, neither object code nor implementation interfaces will be generated.

4.3 The Runtime

To execute any compiled program the runtime, "x", is called upon. Only program modules created by the program compiler can be run. The runtime can take no arguments; the filename must be supplied only when the runtime requests it. The runtime will ask whether the user wants to use the trace option. If the user responds "y" then the runtime will display each line of object code in hexadecimal and octal. Unless the user understands the internal object code, this option ought not be used. When the runtime asks for the filename two options can be used. Either the full name can be given, or the name can be given without the suffix following the period.

Example:

Consider the stack manipulation example as it was first given. A carriage return follows every command.

First, the Parameters definition module must be created:

```
DEFINITION MODULE Parameters;

CONST
    MAxStackSize = 10;
TYPE
    StackType = INTEGER;  (* something simple *)

END Parameters.
```

Now, Parameters can be compiled with the definition compiler.

Type in:

```
d Parameters.def
or
d Parameters
```

The computer will respond with:

```
Parameters.def
Modula-2
in>
 + Parameters.SBL.
```

Everything is now prepared for StackManipulation's definition module to be compiled.

Type in:

```
d StackManipulation.def
or
d StackManipulation
```

The computer will respond with:

```
StackManipulation.def
Modula-2
in>
 Parameters: Parameters.SBL

 + StackManipulation.SBL.
```

The last action the user must perform is compiling the implementation module. The program compiler is used for this.

Type in:

```
c StackManipulation.mod
or
c StackManipulation
```

The computer will respond:

```
StackManipulation.mod
Modula-2
in>
 StackManipulation: StackManipulation.SBL

 Parameters: Parameters.SBL

 + StackManipulation.RFC....
 + StackManipulation.OBJ 113
```

The first things listed are all the files the module wishes to import. The last things listed are the new files the compiler created and the size of these files if appropriate.

StackManipulation is now ready to be used in any program the user creates. If the user wishes to change either StackType of MAxStackSize, Parameters must be edited and all three files must be compiled again.

Whenever a program requests the use of an imported module, the runtime must bring that module into memory. The first time the runtime encounters a reference to an imported module, the runtime will fetch its object code into memory and execute its initialization statements if it has any. This is called *dynamic linking*. The object code must be located in the current directory or else the runtime will not be able to find it. The initialization sequence is only performed once. If many modules import the same module, that module's initialization code will only be performed when it is first read in.

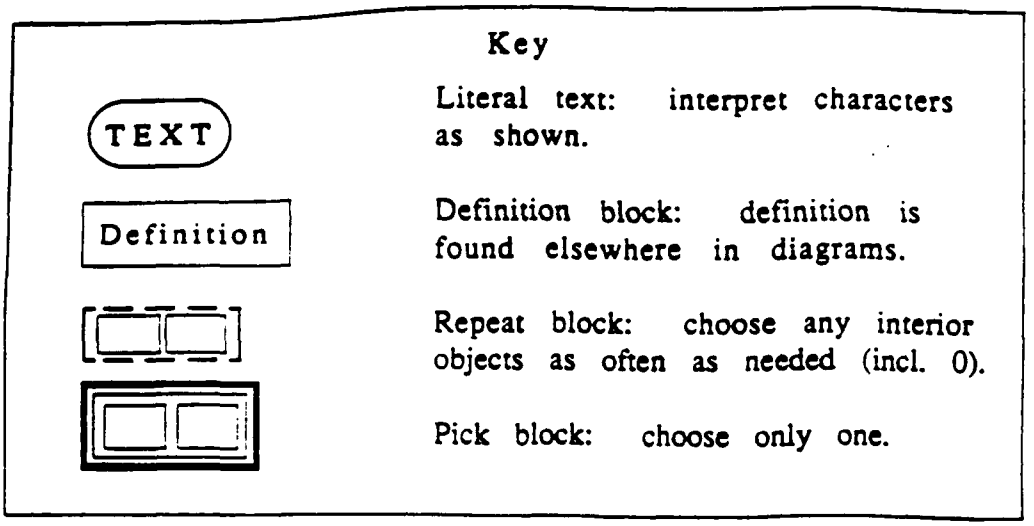
If the runtime encounters an error, a message will be printed. Since all runtime errors are fatal to the program, execution will immediately stop. These are all the possible error messages that the runtime can issue:

normal exit
HALT statement
CASE error
stack overflow
heap overflow
missing RETURN in a function
address error
REAL overflow
REAL underflow
bad operand
CARDINAL overflow
INTEGER overflow
subrange or subscript error
division by zero
illegal instruction
breakpoint

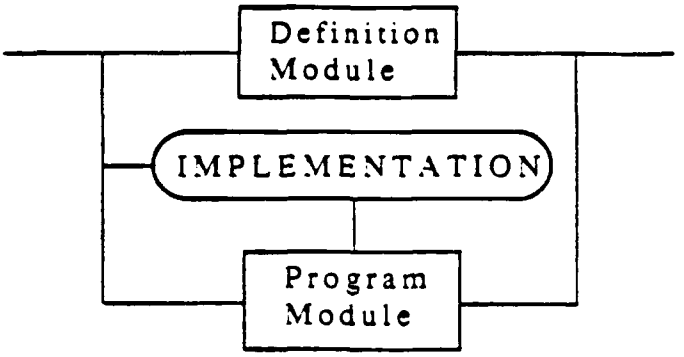
REFERENCES

[1] Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (Dec. 1972) 1053-1058.

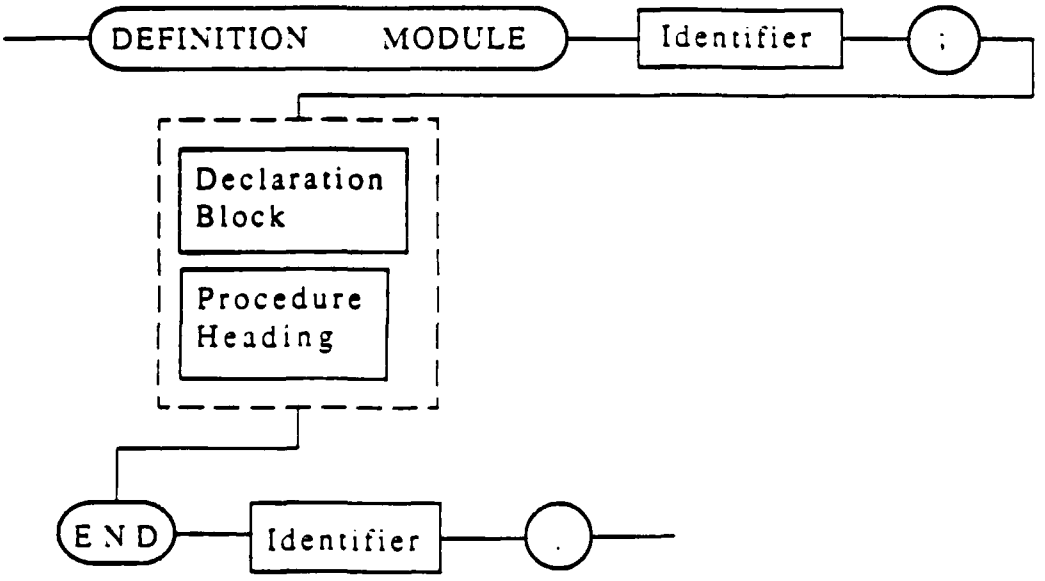
Modula-2 Syntax Diagrams



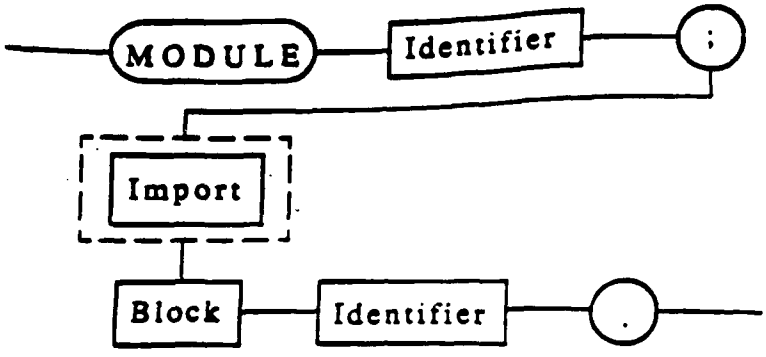
Compilation Unit



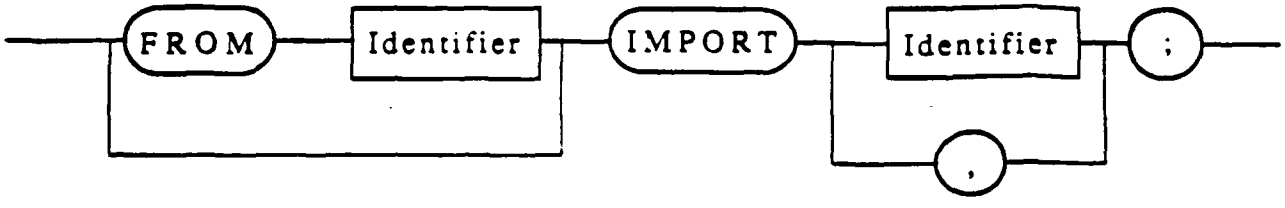
Definition Module



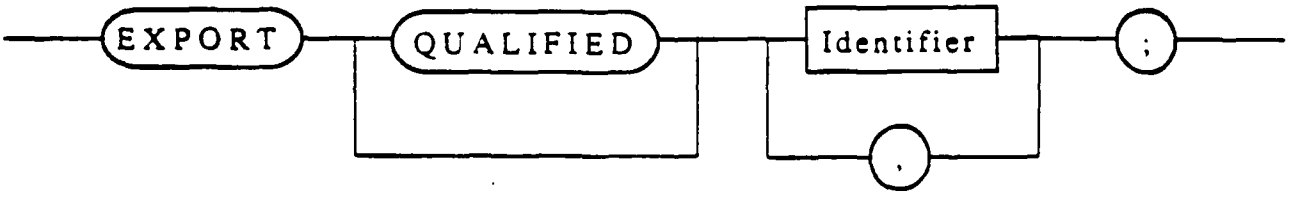
Program Module



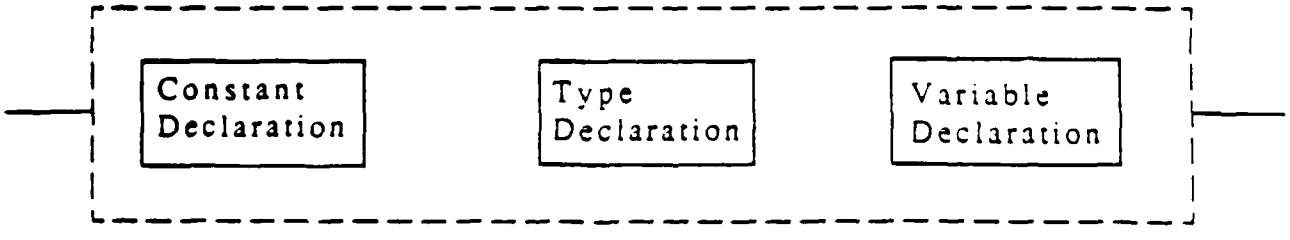
Import



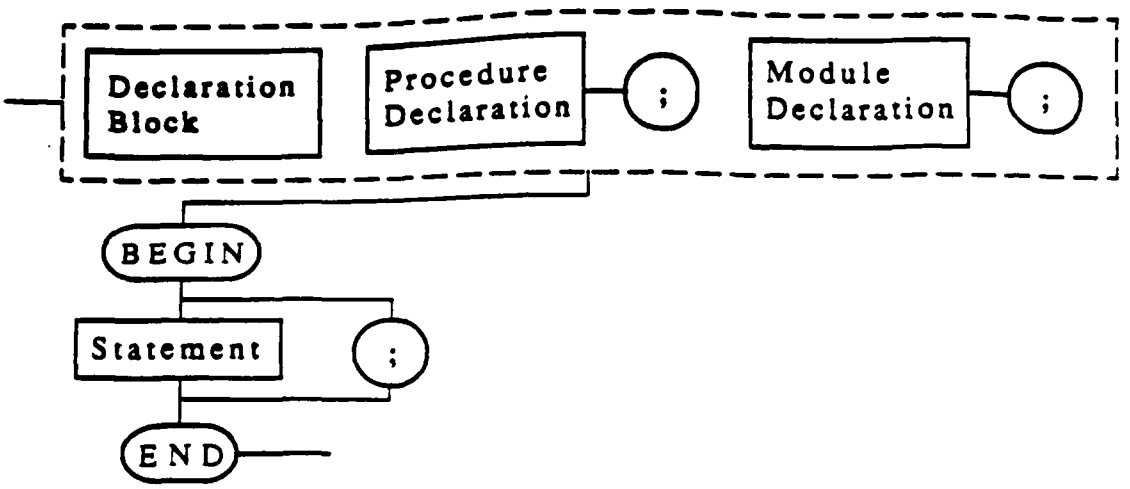
Export



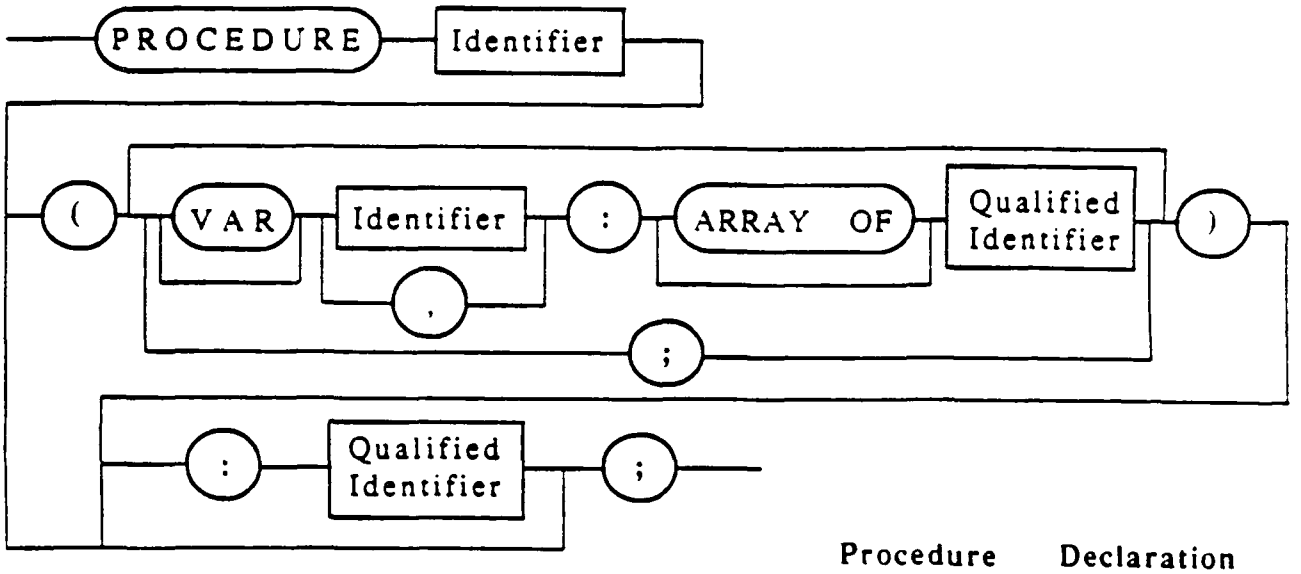
Declaration Block



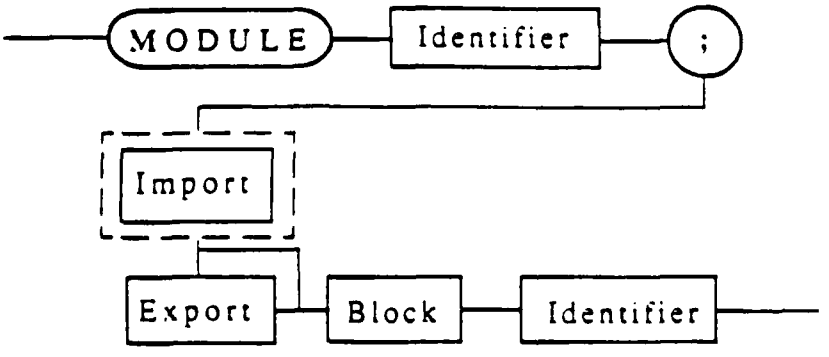
Block



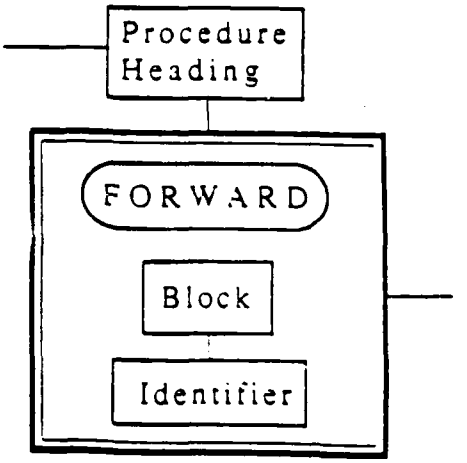
Procedure Heading



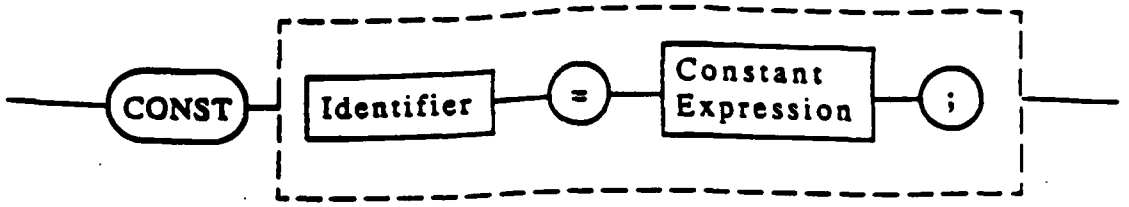
Module Declaration



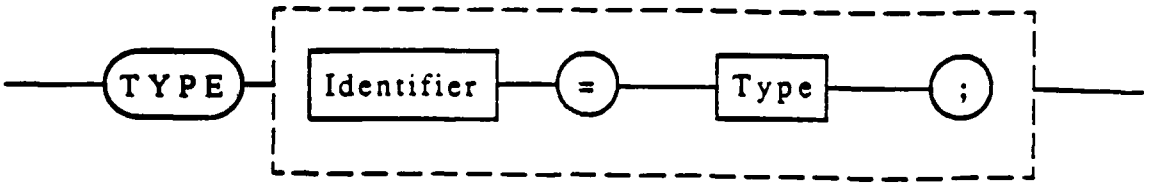
Procedure Declaration



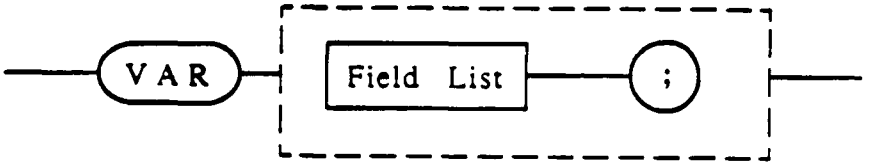
Constant Declaration



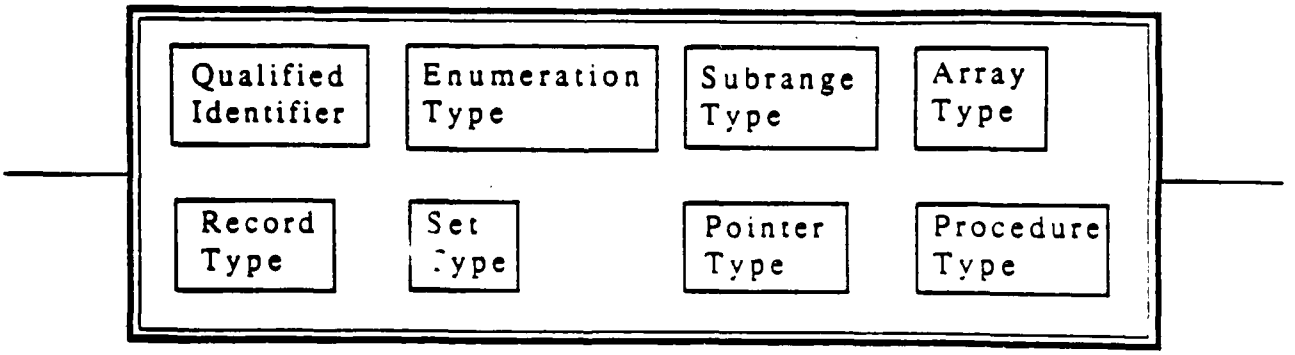
Type Declaration



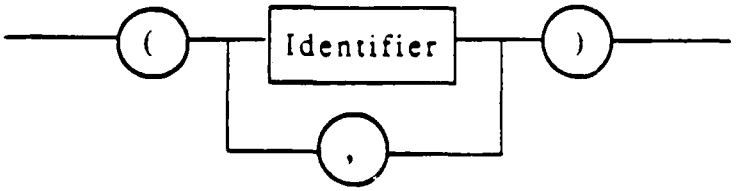
Variable Declaration



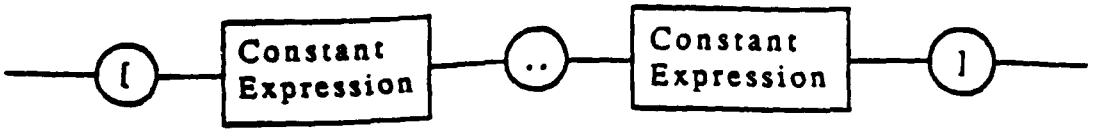
Type



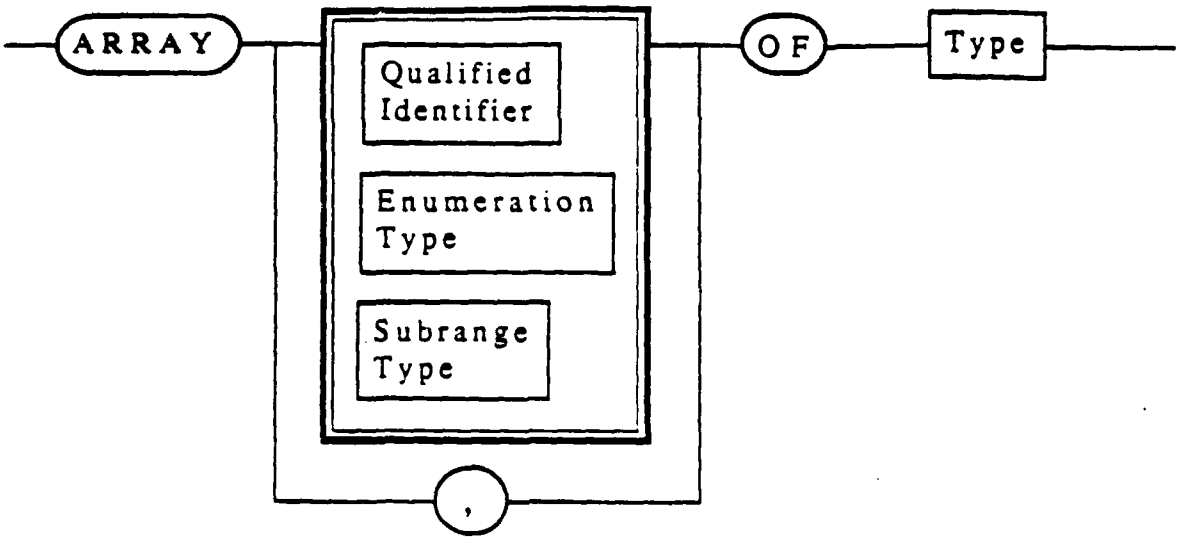
Enumeration Type



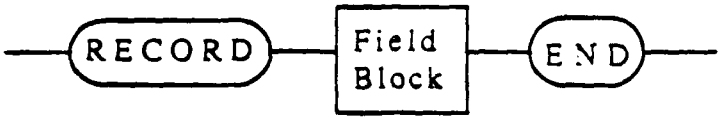
Subrange Type



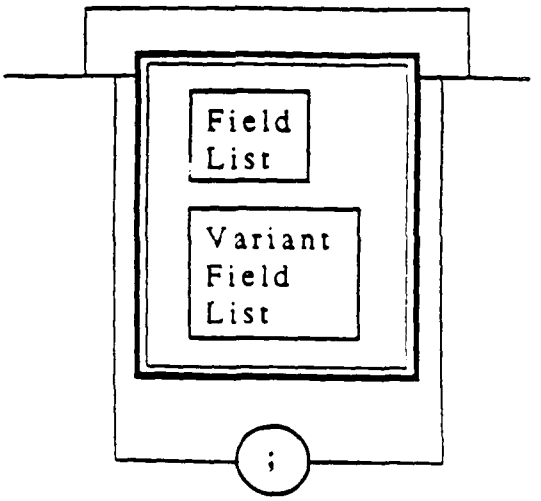
Array Type



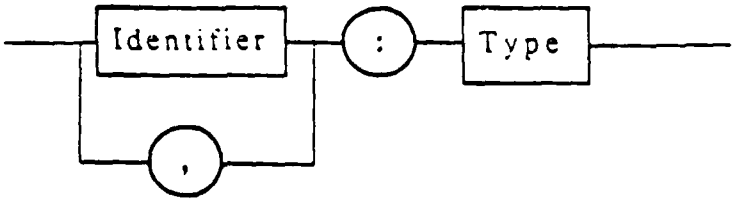
Record Type



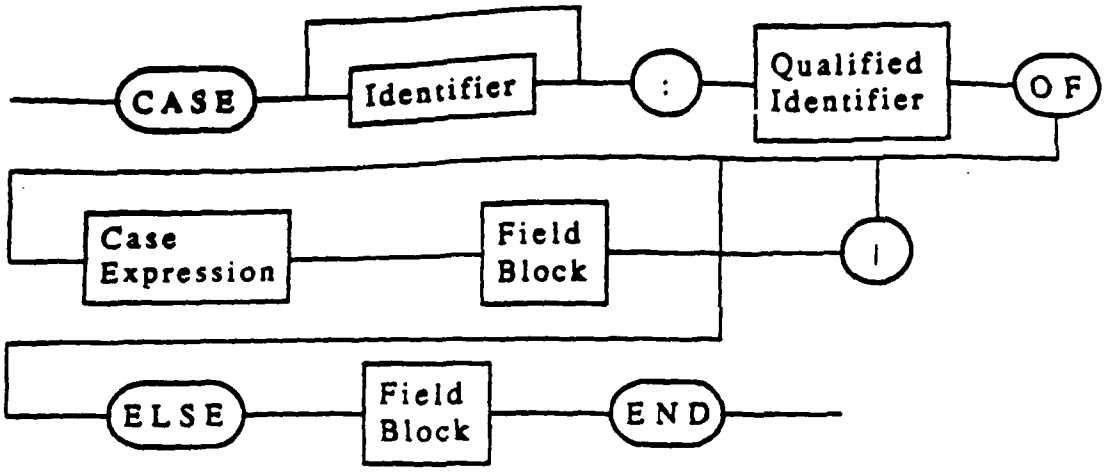
Field Block



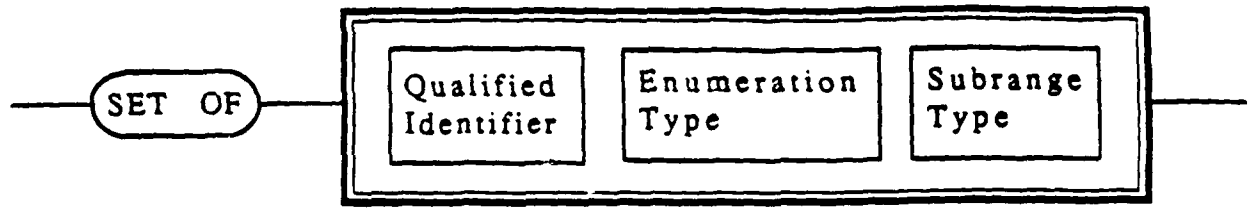
Field List



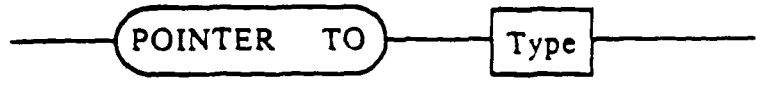
Variant Field List



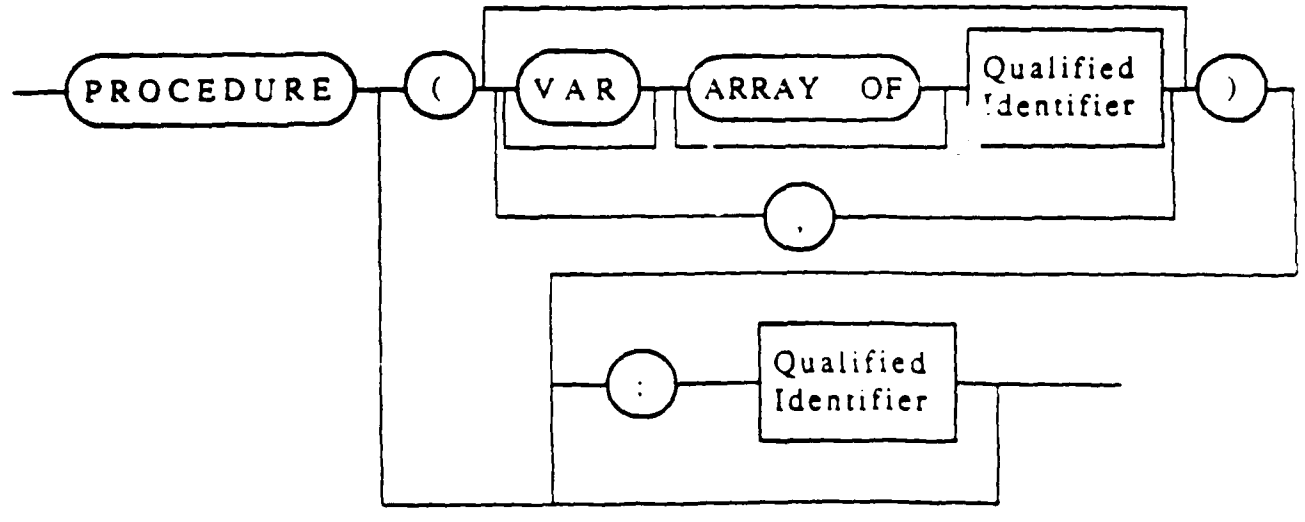
Set Type



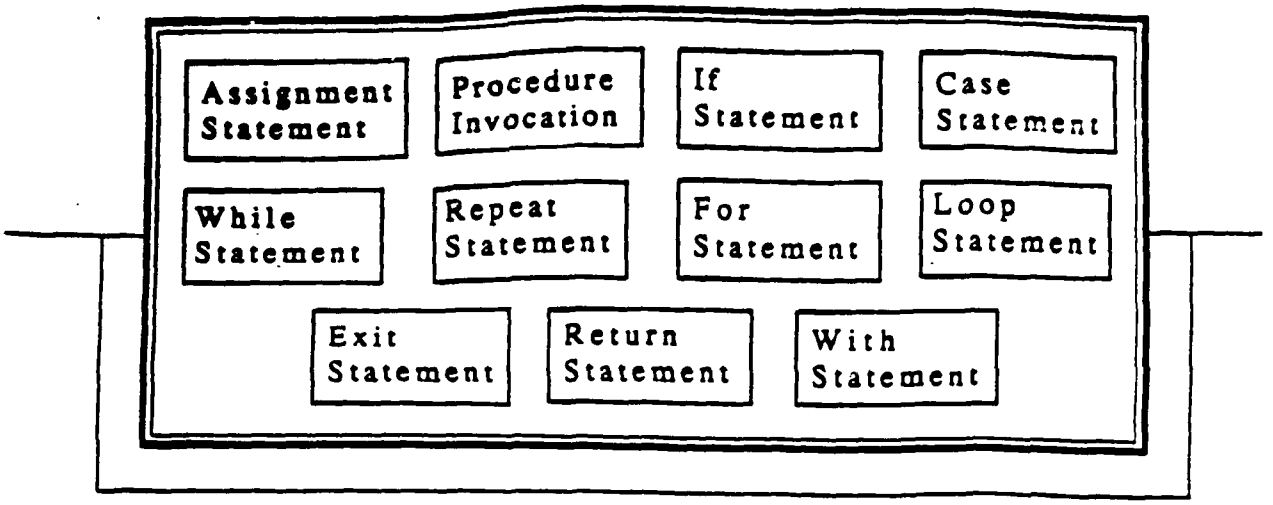
Pointer Type



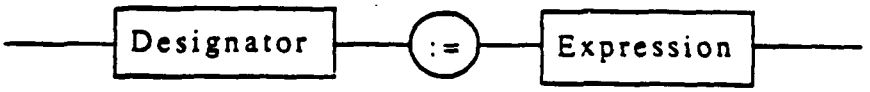
Procedure Type



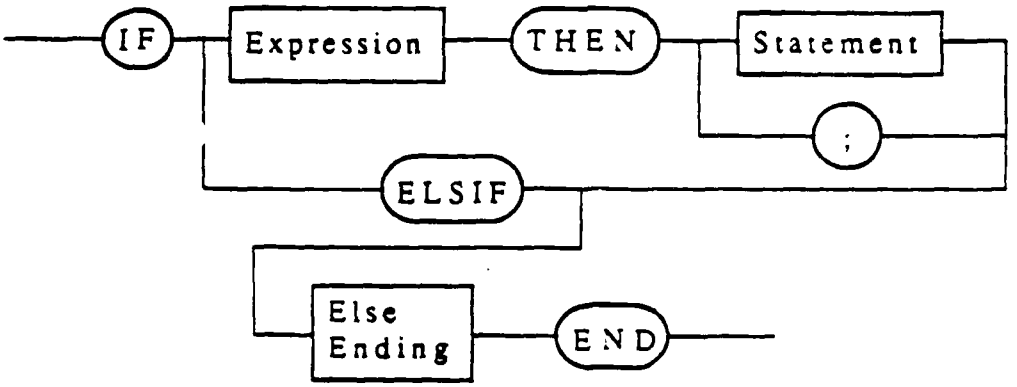
Statement



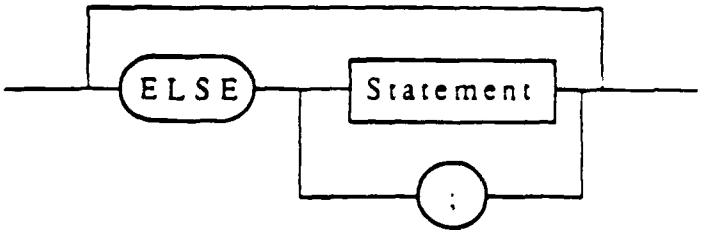
Assignment Statement



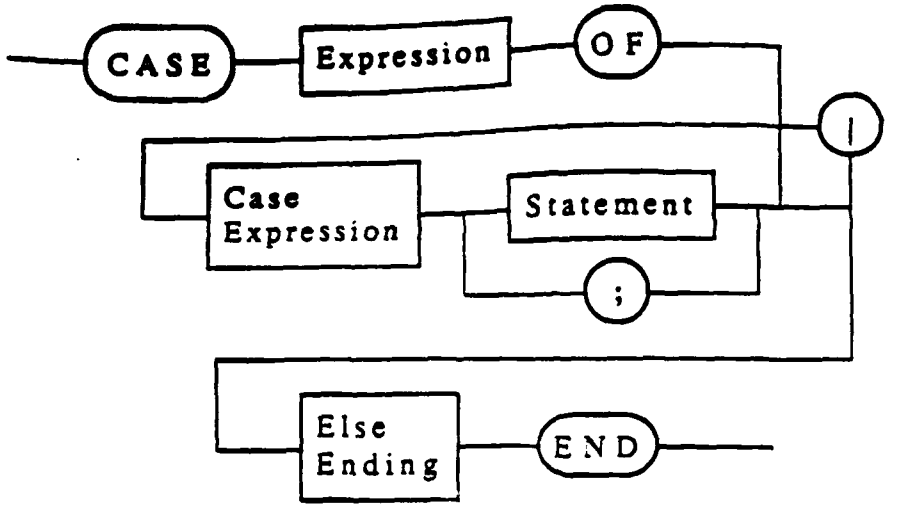
If Statement



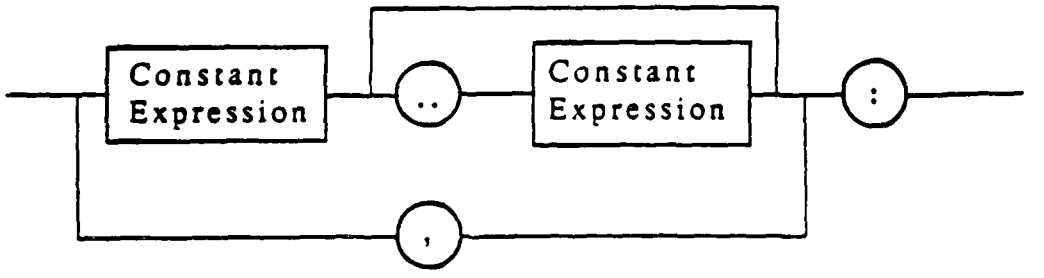
Else Ending



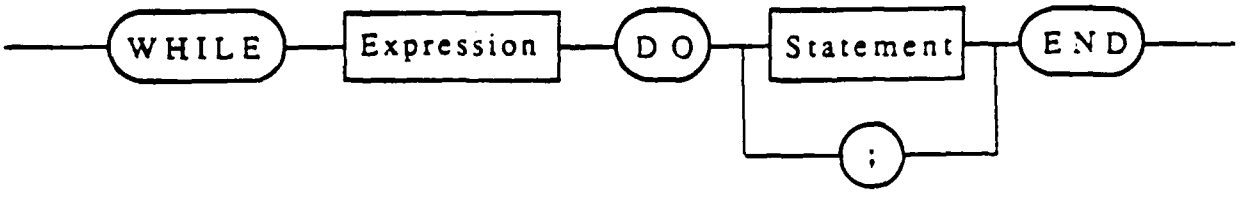
Case Statement



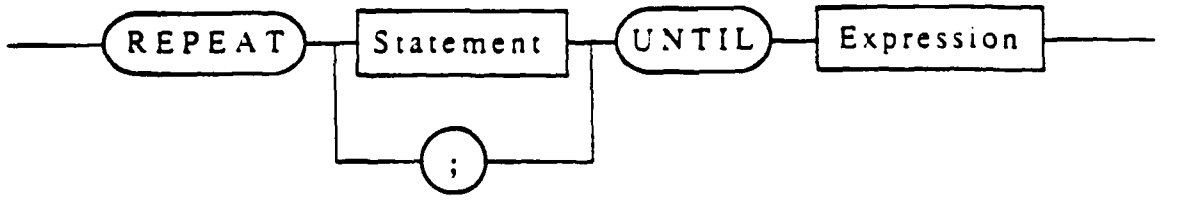
Case Expression



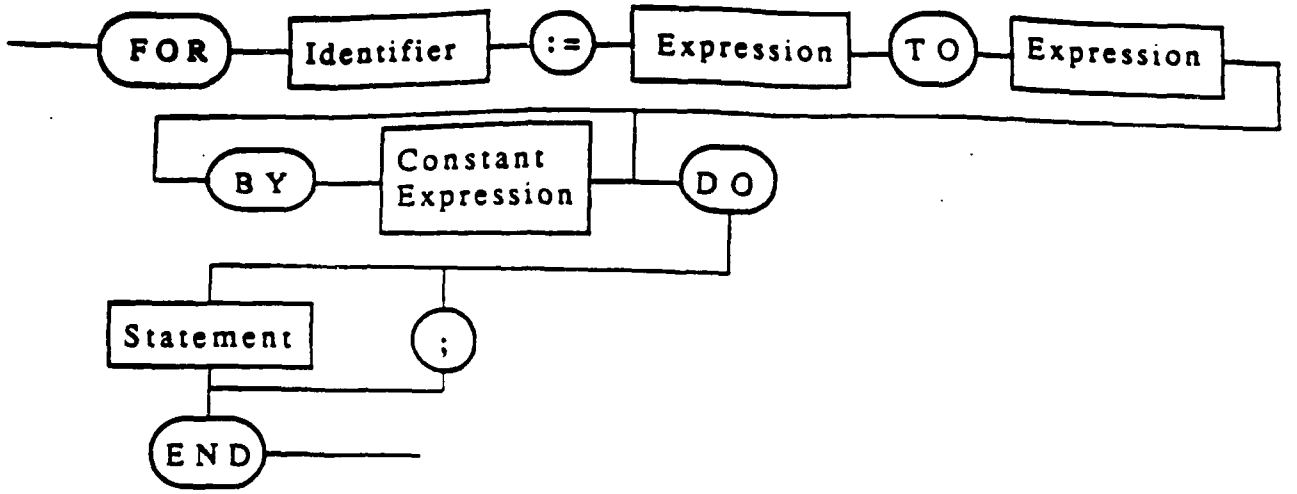
While Statement



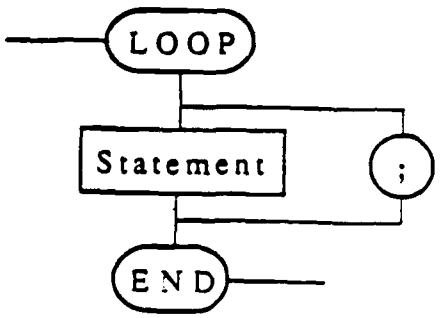
Repeat Statement



For Statement

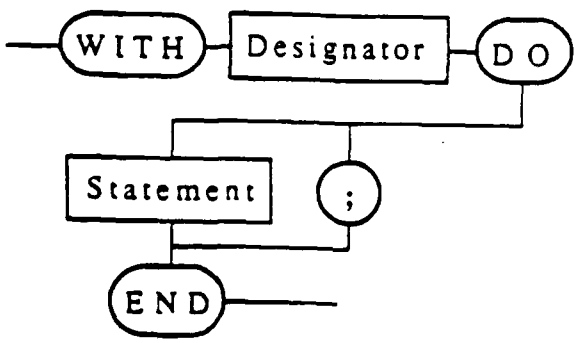


Loop Statement

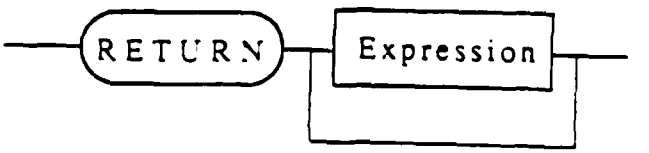


Exit Statement

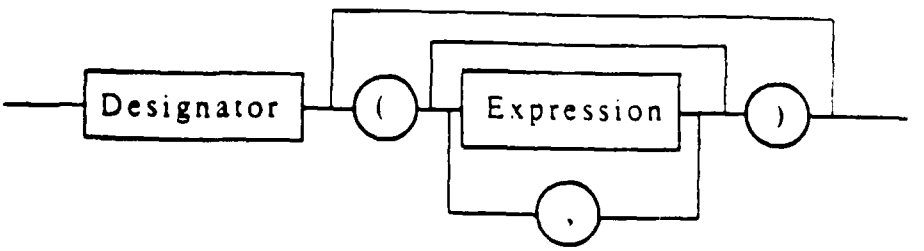
With Statement



Return Statement

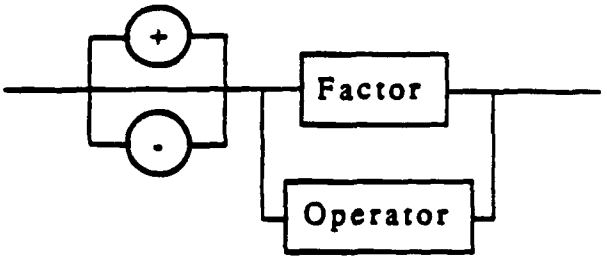


Procedure Invocation

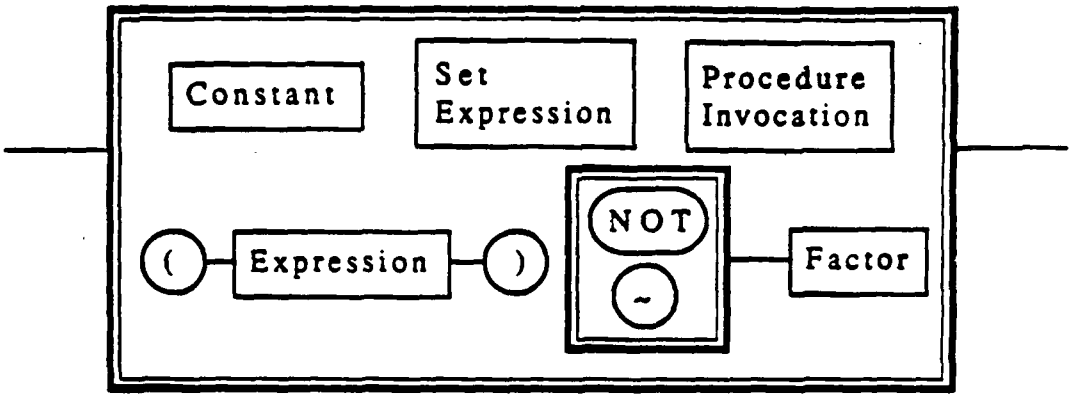


Expression

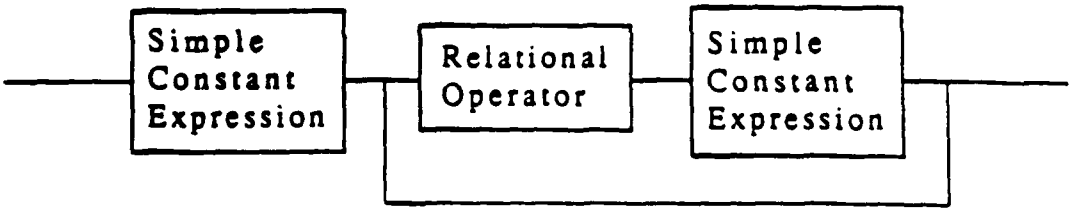
Simple Expression



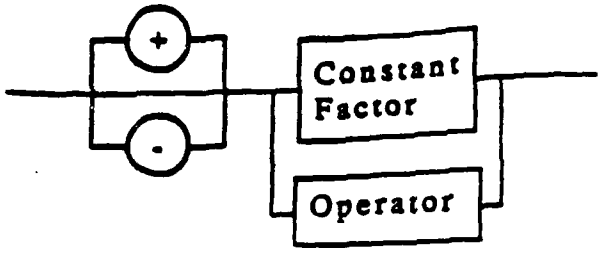
Factor



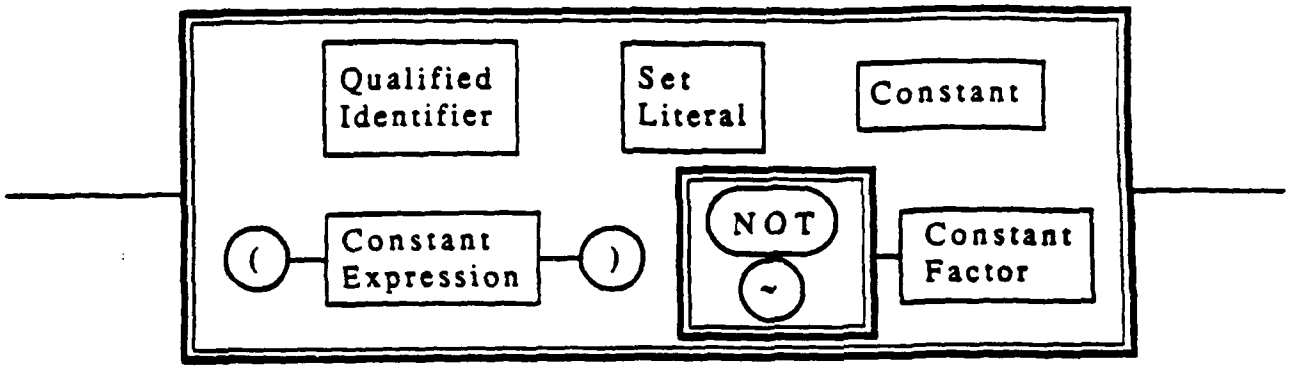
Constant Expression



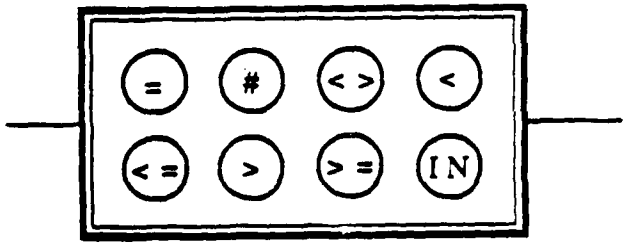
Simple Constant Expression



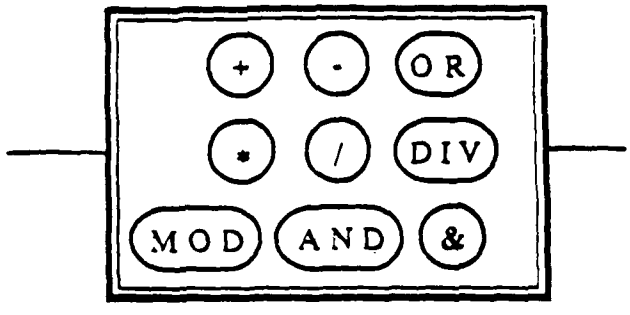
Constant Factor



Relational Operator

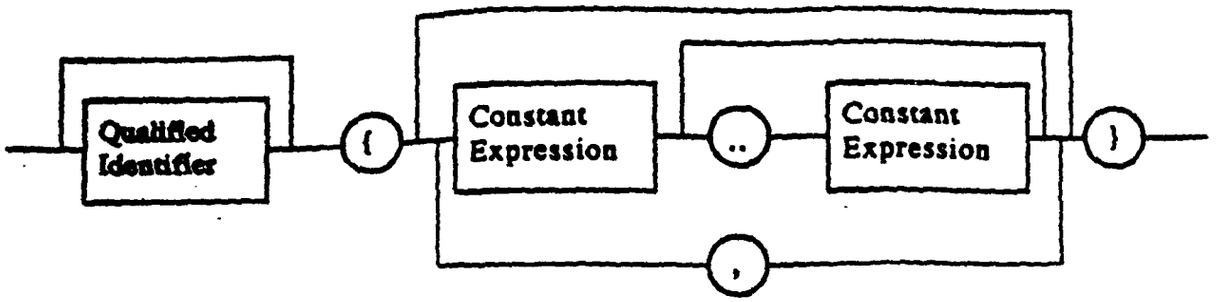


Operator

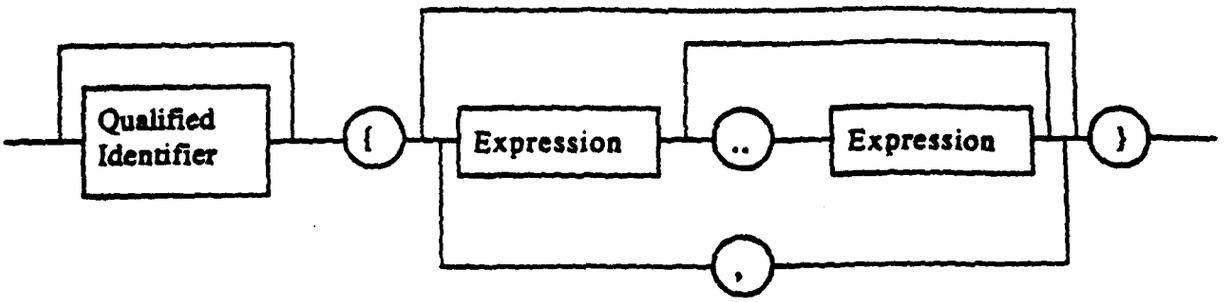


Precedence *highest to lowest*	
NOT, ~, +, -	logical not, unary plus, minus
***, /, MOD DIV, AND, &**	multiplication operators
+, -, OR	addition operators
=, #, <>, < <=, >, >=, IN	relational operators

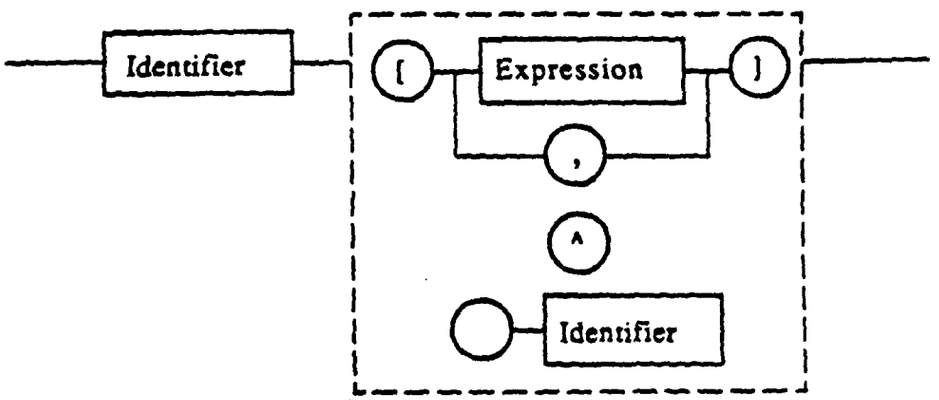
Set Literal



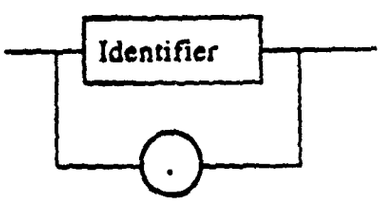
Set Expression



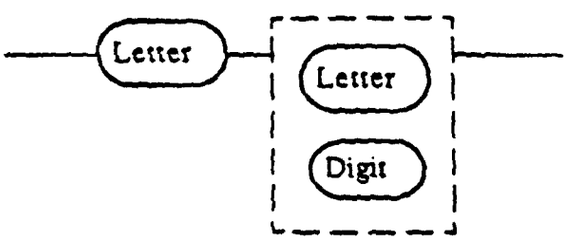
Designator



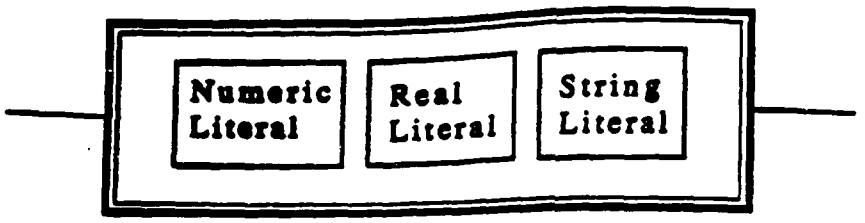
Qualified Identifier



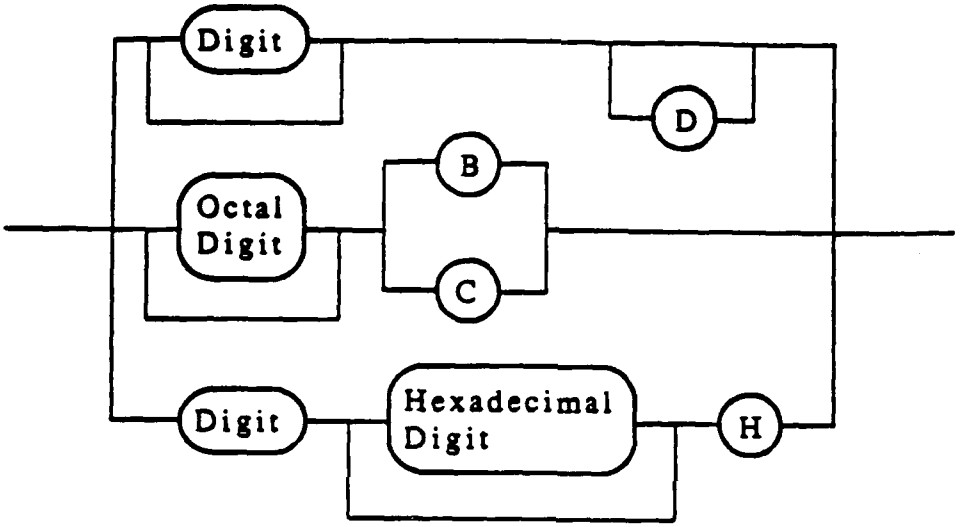
Identifier



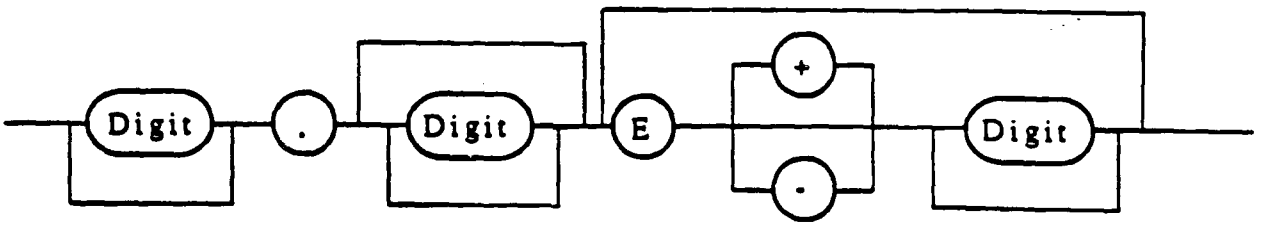
Constant



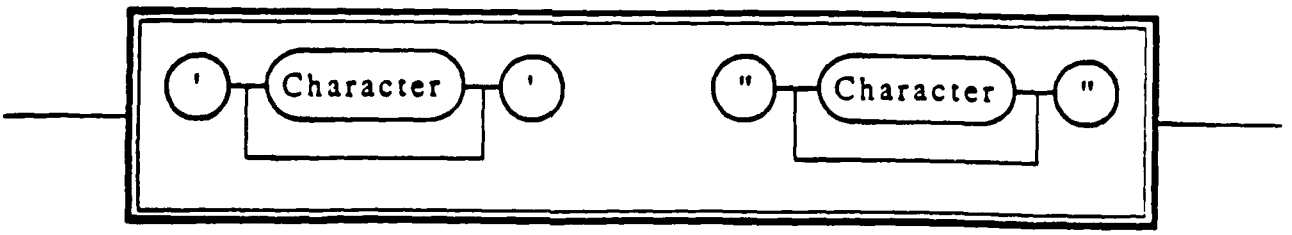
Numeric Literal



Real Literal



String Literal



DISTRIBUTION LIST

Copy No.	
1 - 50	U.S. Army Research Office P.O. Box 12211 4300 S. Miami Boulevard Research Triangle Park, NC 27709-2211
51	Dr. David W. Hislop Electronics Division U.S. Army Research Office P.O. Box 12211 4300 S. Miami Boulevard Research Triangle Park, NC 27709-2211
52 - 53	Dr. R. P. Cook, CS
54	Dr. A. K. Jones, CS
55 - 56	Ms. E. H. Pancake, Clark Hall
57	SEAS Publications Files
*	Office of Naval Research Resident Representative 818 Connecticut Ave., N.W. Eighth Floor Washington, DC 20006 Attention: Mr. Michael McCracken Administrative Contracting Officer

2157:ald:DW213R

UNIVERSITY OF VIRGINIA
School of Engineering and Applied Science

The University of Virginia's School of Engineering and Applied Science has an undergraduate enrollment of approximately 1,500 students with a graduate enrollment of approximately 560. There are 150 faculty members, a majority of whom conduct research in addition to teaching.

Research is a vital part of the educational program and interests parallel academic specialties. These range from the classical engineering disciplines of Chemical, Civil, Electrical, and Mechanical and Aerospace to newer, more specialized fields of Biomedical Engineering, Systems Engineering, Materials Science, Nuclear Engineering and Engineering Physics, Applied Mathematics and Computer Science. Within these disciplines there are well equipped laboratories for conducting highly specialized research. All departments offer the doctorate; Biomedical and Materials Science grant only graduate degrees. In addition, courses in the humanities are offered within the School.

The University of Virginia (which includes approximately 2,000 faculty and a total of full-time student enrollment of about 16,400), also offers professional degrees under the schools of Architecture, Law, Medicine, Nursing, Commerce, Business Administration, and Education. In addition, the College of Arts and Sciences houses departments of Mathematics, Physics, Chemistry and others relevant to the engineering research program. The School of Engineering and Applied Science is an integral part of this University community which provides opportunities for interdisciplinary work in pursuit of the basic goals of education, research, and public service.